2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

Data Compression using Artificial Neural Networks

by

Bruce E. Watkins

September 1991

Thesis Advisor: Murali Tummala

Approved for public release; distribution is unlimited

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE — Form Approved OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School — 6b OFFICE SYMBOL (If applicable) Code 32	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION — 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO — PROJECT NO — TASK NO — WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
DATA COMPRESSION USING ARTIFICIAL NEURAL NETWORKS

12 PERSONAL AUTHOR(S) Watkins, Bruce E.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Engineer's Thesis	FROM ______ TO ______	1991 September	93

16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Neural Networks, Vector Quantization, Image Coding

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis investigates the application of artificial neural networks for the compression of image data. An algorithm is developed using the competitive learning paradigm which takes advantage of the parallel processing and classification capability of neural networks to produce an efficient implementation of vector quantization. Multi-Stage, tree searched, and classification vector quantization codebook design are adapted to the neural network design to reduce the computational cost and hardware requirements. The results show that the new algorithm provides a substantial reduction in computational costs and an improvement in performance.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Tummala, Murali	22b TELEPHONE (Include Area Code) 408-646-2645 — 22c OFFICE SYMBOL Code EC/Tu

DD Form 1473, JUN 86 — Previous editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE
S/N 0102-LF-014-6603 — Unclassified

Data Compression Using Artificial Neural Networks

by

Bruce E. Watkins
Lieutenant, USN
B.S. University of California, Santa Barbara, 1984

Submitted in partial fulfillment of the
requirements for the degree of

ELECTRICAL ENGINEER

from the

NAVAL POSTGRADUATE SCHOOL

September, 1991

Author: [illegible]
Bruce E. Watkins

Approved by: [illegible]
Murali Tummala, Thesis Advisor

Charles W. Therrien
Charles W. Therrien, Second Reader

Michael A. Morgan
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

[illegible]
Richard S. Elster, Dean of Instruction

ABSTRACT

This thesis investigates the application of artificial neural networks for the compression of image data. An algorithm is developed using the competitive learning paradigm which takes advantage of the parallel processing and classification capability of neural networks to produce an efficient implementation of vector quantization. Multi-Stage, tree searched, and classification vector quantization codebook design techniques are adapted to the neural network design to reduce the computational cost and hardware requirements. The results show that the new algorithm provides a substantial reduction in computational costs and an improvement in performance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Despite all the recent advances in the areas of communications and data coding, there are still a large number of applications where the achievable data rate is not sufficient to the task. There also exists an even larger number of tasks for which an improvement in data compression would enable us to do the job better or more efficiently. Two prime examples of the types of data for which better coding is desirable are digital speech and image data. Both of these data types require an extremely high data rate for real time transmission. Both also display a wealth of internal structure that can be utilized for compression by a coding system. Finally both of these types of signals can be transmitted with a certain amount of distortion and still provide the required information. For example, it may be sufficient to maintain intelligibility for speech data, and it may be sufficient for an image to display enough detail for an analyst to recognize certain key features rather than a faithful bit by bit reproduction. This is in contrast to many other types of digital data for which our principle interest is to add error correcting capability until the probability of a single bit error is vanishingly small. All these factors combine to make improved compression techniques for speech and image data a worthy goal and thus an active area of research in digital signal processing.

In signals for which we can tolerate some distortion, there must be some method for measuring the distortion relative to the original signal. These fall into the two basic categories of subjective distortion measures and objective distortion measures. The subjective measures are a result of human impressions of the comparison between original and distorted versions; while the objective measure has some closed form mathematical expression by which we can compare competing systems. For our study we desire a data type that has a simple objective measure that corresponds well to the results of subjective measures. Fortunately, for image data there exists a distortion measure, mean square error, which is both easy to calculate and corresponds fairly well to subjective distortion results. Thus in this thesis, we concentrate on the compression of image data.

There are many schemes for compressing image data, but few have been successful in producing good images quality at low data rates. Generally, images are coded with each pixel assigned a grey level from 0 to 255. This corresponds to a data rate of 8 bits/pixel. Several examples showing how three common coding techniques perform at low data rates are shown in the following figures. First, we examine the technique of scalar quantization, in which we map the 256 grey levels into a smaller number, which can then be transmitted using a smaller number of bits. Figure 1.1 shows the original at 8 bits/pixel, Figure 1.2 shows scalar quantization at a data rate of 4 bits/pixel, Figure 1.3 shows a data rate of 2 bits /pixel, and figure 1.4 shows a data rate of 1 bit/pixel. Clearly this technique produces poor results below about 4 bits/pixel. Second, we examine the technique of delta modulation, in which we encode the difference between the current and previous pixels using a raster scan. Figure 1.5 shows the original image, Figure 1.6 shows delta modulation at a data rate of 4 bits/pixel, Figure 1.7 shows a data rate of 2 bits/pixel, and Figure 1.8 shows a data rate of 1 bit/pixel. While delta modulation is an improvement over scalar quantization, it tends to perform poorly below about 2 bits/pixel. Next, we examine a transform technique, the two dimensional fast fourier transform (2-D FFT). Figure 1.9 shows the original image, Figure 1.10 shows the 2-D FFT at a data rate of 4 bits/pixel, Figure 1.11 shows a data rate of 2 bits/pixel, and Figure 1.12 shows a data rate of 1 bit/pixel. This technique offers fairly good reproduction down to 2 bits/pixel.

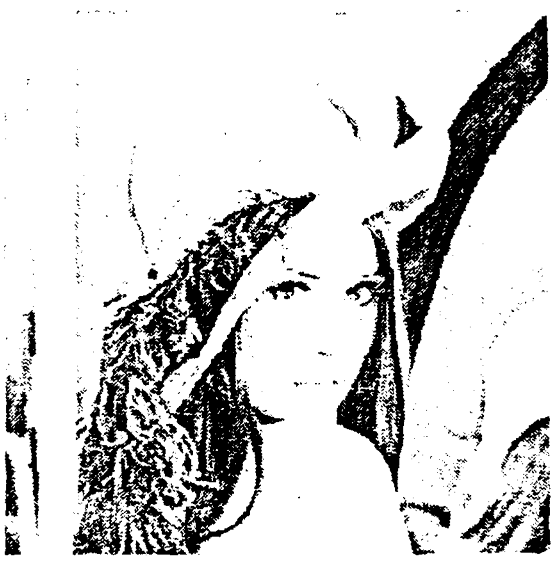
Figure 1.1 Original Image

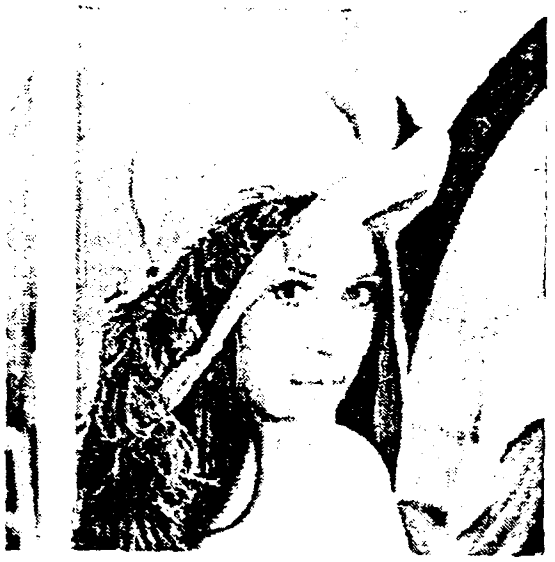
Figure 1.2 Scalar Quantization at 4 bits/pixel

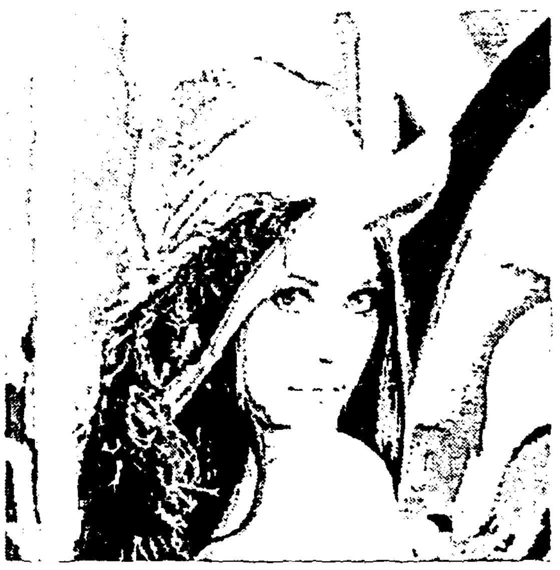
Figure 1.3 Scalar Quantization at 2 bits/pixel

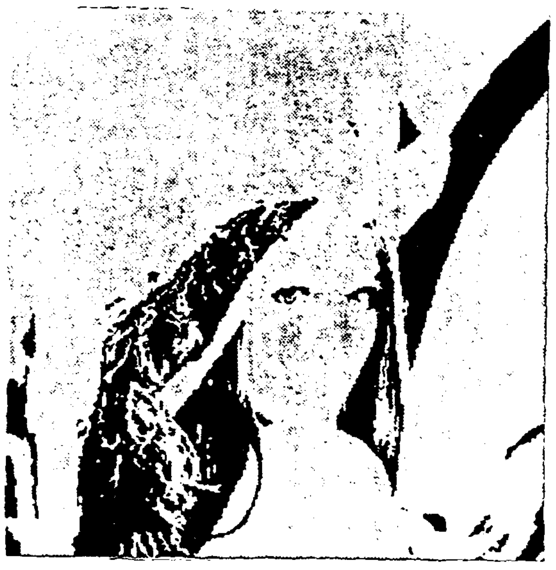
Figure 1.4 Scalar Quantization at 1 bit/pixel

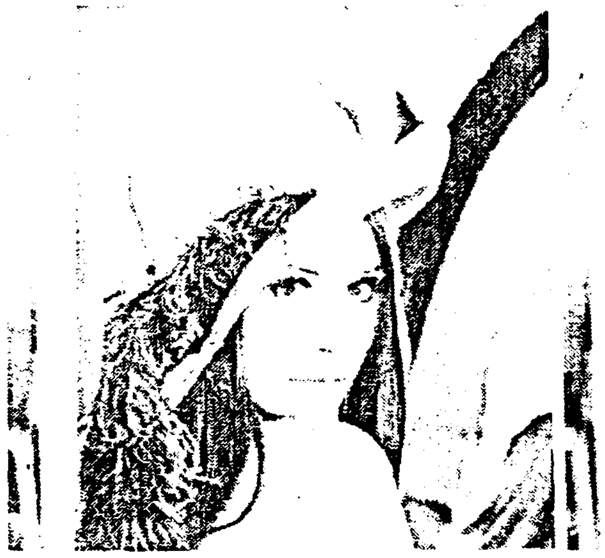

Figure 1.5 Original Image

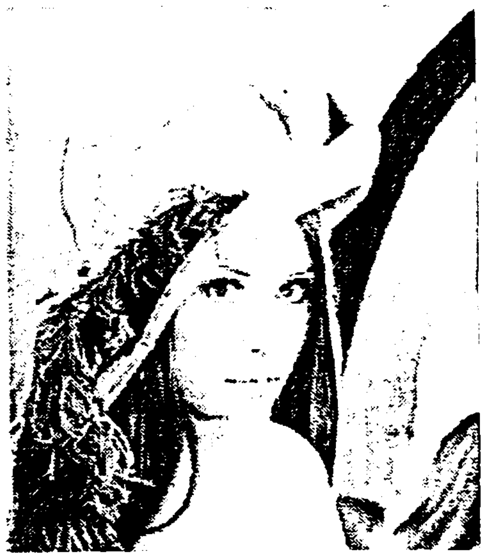

Figure 1.6 Delta Modulation
at 4 bits/pixel

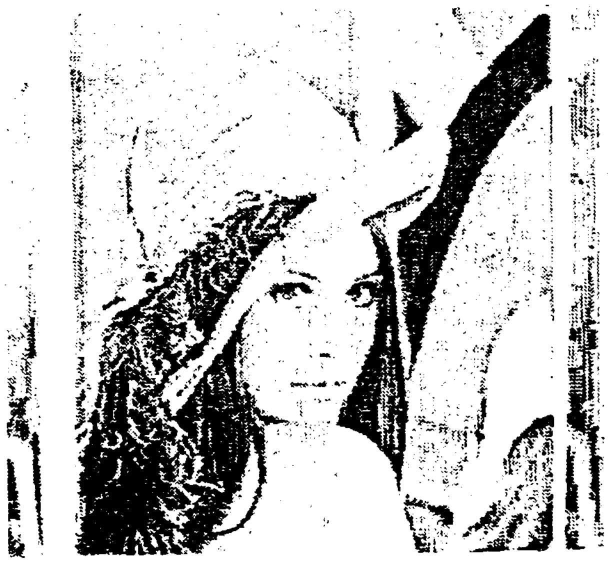

Figure 1.7 Delta Modulation
at 2 bits/pixel

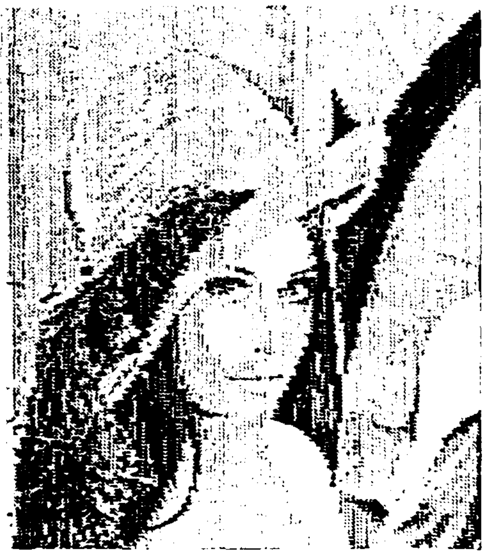

Figure 1.8 Delta Modulation
at 1 bit/pixel

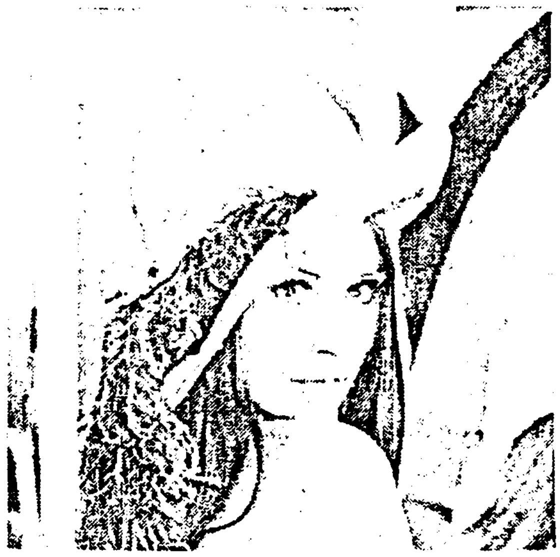

Figure 1.9 Original Image

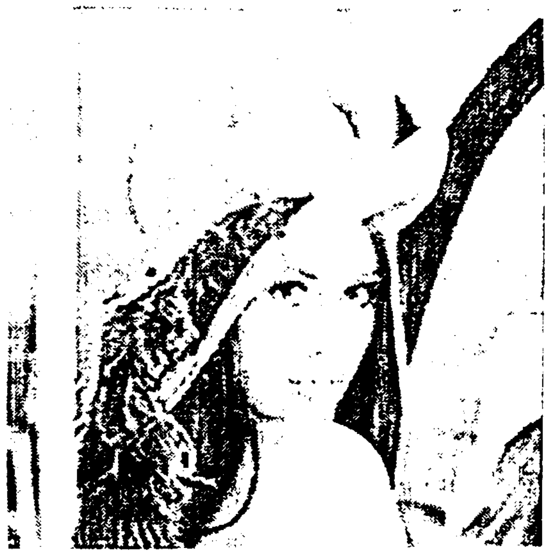

Figure 1.10 2-D FFT at 4 bits/pixel

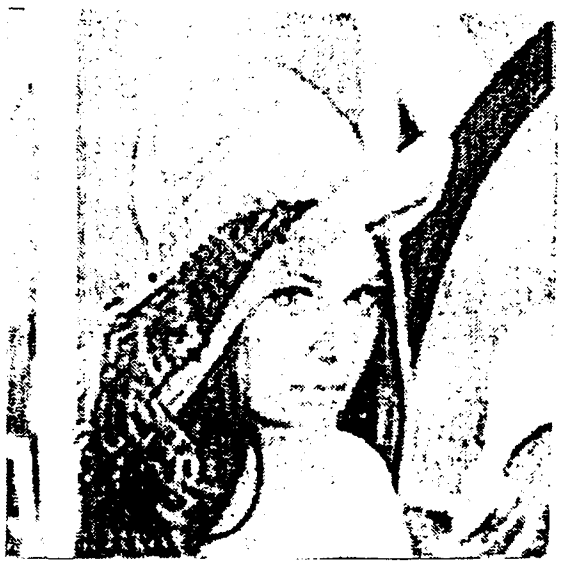

Figure 1.11 2-D FFT at 2 bits/pixel

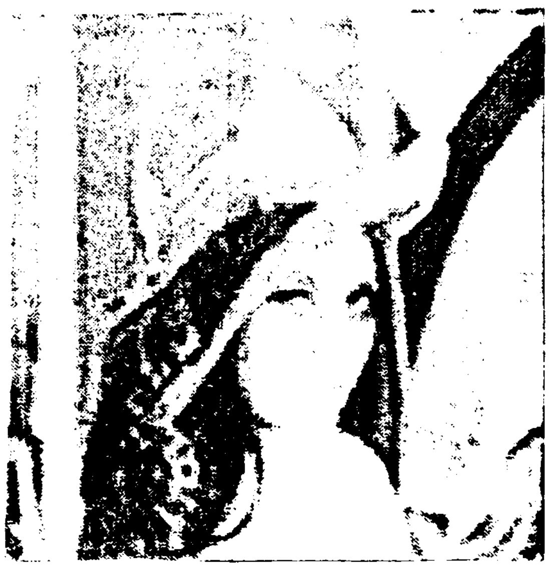

Figure 1.12 2-D FFT at 1 bit/pixel

Now we compare the previous techniques to a more powerful method, vector quantization. In vector quantization, we encode an entire block of data using a single codeword. This codeword is produced by comparing the block to be encoded with a codebook of example blocks and choosing the example which is closest in some sense. A comparison of vector quantization and the previous three techniques is made in the following figures at a data rate of 1 bit/pixel. Figure 1.13 shows scalar quantization at 1 bit/pixel, Figure 1.14 shows delta modulation at 1 bit/pixel, Figure 1.15 shows the 2-D FFT at 1 bit/pixel, and Figure 1.16 shows vector quantization at 1 bit/pixel. Clearly the technique of vector quantization is superior at low data rates. There exist other coding techiques [Ref. 1] such as linear predictive coding and the discrete cosine transform which have been successful in image coding, but were not considered in these examples for the sake of brevity.

A. THESIS OBJECTIVE

Vector quantization has not been commonly used because of the large computational cost involved in generating the codebook and finding the closest codeword for each block to be transmitted. Recently, a revived interest in research in neural networks has shown some promise for an efficient implementation of vector quantization. In this task we benefit from the neural network's ability to quickly perform categorizations (which accelerates the codebook generation), and also from the parallel processing capability (which speeds the process of comparing an input block to the codebook). This thesis concentrates on addressing the difficulties in implementing vector quantization using neural networks. Full Search, tree search, and multistage VQ schemes are studied for this purpose. Results of the application of these techniques to image data are presented.

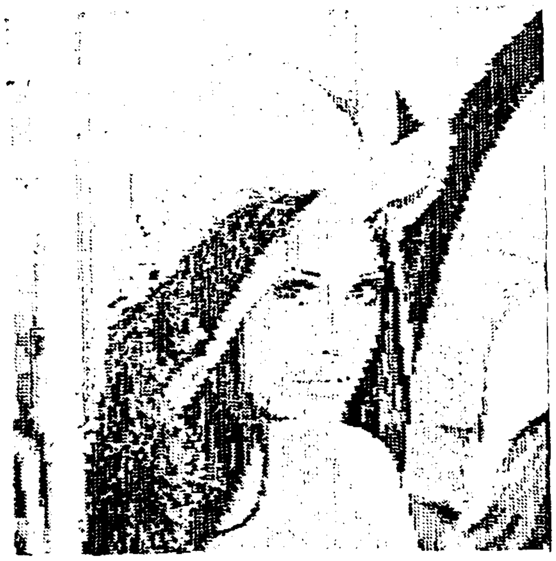

Figure 1.13 Scalar Quantization at 1 bit/pixel

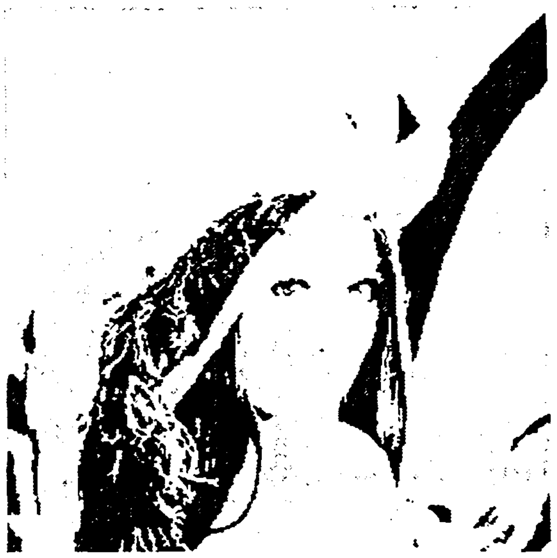

Figure 1.14 Delta Modulation at 1 bit/pixel

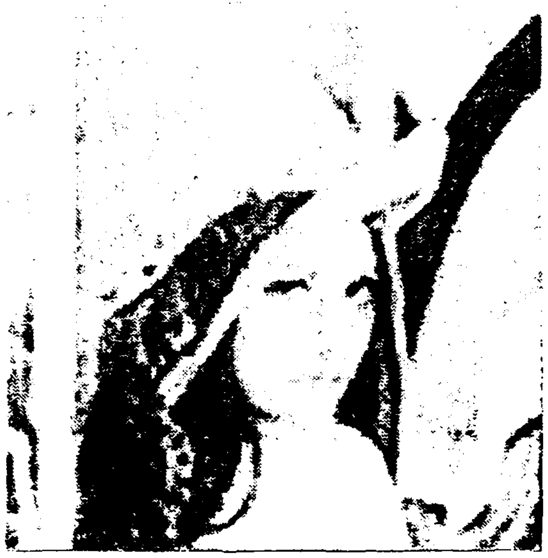

Figure 1.15 2-D FFT at 1 bit/pixel

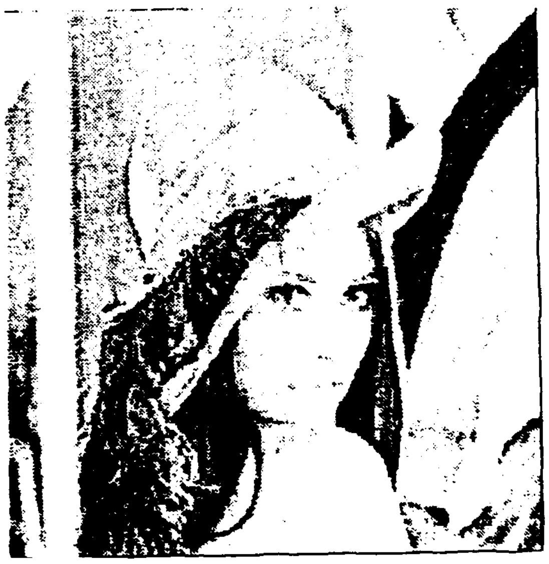

Figure 1.16 Vector Quantization at 1 bit/pixel

B. THESIS OUTLINE

In the second chapter we describe the basic theory of vector quantization, introduce the existing VQ algorithms, and present some simple examples of how a codebook is generated. In the third chapter we introduce the basic concepts of neural networks, discuss the types of neural network learning, and present the algorithms which can be applied to the problem of vector quantization. In the fourth chapter we identify the shortcomings of existing neural network vector quantizers, and apply the tree search, multi stage and classification vector quantization schemes in an effort to improve performance.

II. VECTOR QUANTIZATION

A. INTRODUCTION

One of the results of Shannon's rate-distortion theory [Ref. 3] is that better results can always be obtained if vectors are used in coding rather than scalars. This result applies even if some technique has been applied to the input data to remove all correlation. Although delta modulation and transform methods provide substantial improvement over scalar quantization, they all use scalar coding and are thus suboptimal. As we saw in the examples presented previously, vector quantization provides a dramatic improvement in reproduction quality for low data rates. In this chapter we examine the technique of vector quantization as it applies to images and review existing methods of implementation.

B. DETAILS OF THE METHOD

Vector Quantization (VQ) [Ref. 2] consists of two sets of mappings: an encoder, $\gamma(\mathbf{x})$, which assigns a channel codeword, $\mathbf{u} = (u_1, u_2, \ldots, u_\rho)$, to each input vector, $\mathbf{x} = (x_0, x_1, \ldots, x_{k-1})$, from a set of possible channel symbols called a codebook, and a decoder, $\beta(\mathbf{u})$, which assigns a code vector, $\mathbf{y}$, to each channel codeword. Note that each input vector is just a vector version of a block from the subject image with each pixel value corresponding to an element in the vector. The channel symbols consist of all possible binary ρ-tuples, where the size of the input vector and the channel codeword are in general not the same. The number of possible channel codewords is 2^ρ, and thus the bit rate of the vector quantizer is ρ bits/block or $r = \rho/k$ bits/pixel. It is interesting to note that by properly selecting ρ and k, we can generate any fractional bit rate that we desire. This is in contrast to scalar quantization where we are limited to integer bit rates. Figure 2.1 shows the basic structure of a vector quantizer system.

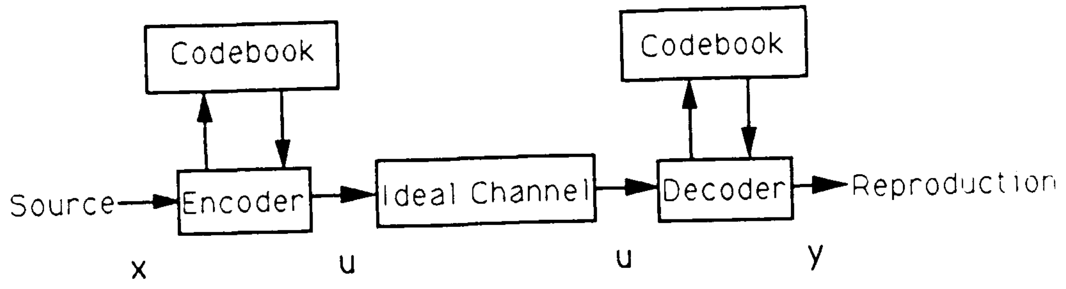

Figure 2.1: Vector Quantization

The basic goal of the vector quantizer design is to discover which specific set of encoder and decoder mappings will give the best reproduction of the image in some sense. This depends upon the cost function or distortion which we use to measure the quality of the output image. We wish to find a distortion or distance measure between our input image, $\mathbf{X}$, and the output image, $\hat{\mathbf{X}}$, that is easy to compute and provides good correspondence with subjective image quality. The measure chosen for this work is the mean square error (MSE) which is defined as

$$\epsilon = d(\mathbf{X},\hat{\mathbf{X}}) = \frac{1}{N^2}\|\mathbf{X}-\hat{\mathbf{X}}\|^2 = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\mathbf{x}_{ij}-\hat{\mathbf{x}}_{ij})^2 \qquad (2.1)$$

where N is the number of pixels along one side of the image.

We will now examine the conditions under which the vector quantizer will be optimal. First we define the set of all possible code vectors, $\mathbf{y}$, as $\mathbf{C} = [\mathbf{y} : \forall\ \mathbf{y} \in \beta(\mathrm{u})]$. This set of code vectors together with the corresponding codewords is called the codebook. For the vector quantizer to be optimal it must display two properties [Ref. 5].

First, the encoder must select the code vector which is closest to the input vector according to the distortion measure. In our case this is the mean square error. This can be stated as

$$d(\mathbf{x}, \beta[\gamma(\mathbf{x})]) = \min_{\mathbf{u}} d[\mathbf{x}, \beta(\mathbf{u})] = \min_{\mathbf{y} \in \mathbf{C}} d(\mathbf{x}, \mathbf{y}). \tag{2.2}$$

Thus the encoder can be thought of as a device which partitions the input vector space into sections which surround a code vector. Any input which falls into that section will be assigned the codeword corresponding the code vector contained in that section. The encoder can also be viewed as a device which divides the input vector space into a group of sections for which all input vectors occuring in a section are grouped together and transmitted as a single representative code vector.

Second, for an encoder γ, the decoder must assign as the code vector the generalized centroid of all the vectors which are encoded into that code word. In our case the centroid can be expressed

$$\mathbf{y} = \beta(\mathbf{u}) = \text{cent}(\mathbf{u}) = \frac{1}{i(\mathbf{u})} \sum_{\mathbf{x}_i : \gamma(\mathbf{x}_i) = \mathbf{u}} \mathbf{x}_i \tag{2.3}$$

where $i(\mathbf{u})$ is the number of input vectors that are mapped to $\mathbf{u}$. This is the selection of the code vector which will minimize the distortion, $E[d(\mathbf{x}, \mathbf{y}) \mid \gamma(\mathbf{x}) = \mathbf{u}]$ for a particular encoder.

If we carefully examine the previous two properties, we can see that the first gives us a method to optimize an encoder for a given decoder, and the second gives us a method to optimize a decoder for a given encoder. This suggests an iterative technique of applying these two properties successively until convergence is obtained or some desired distortion level is reached. This is in fact the basis for the generalization of Lloyd's optimal scalar quantizer [Ref. 4], which was produced by Linde, Buzo and Gray [Ref. 6] and is referred to as the LBG algorithm. The algorithm consists of the following steps:

- **Step 1** Choose an initial decoder.
- **Step 2** Encode the image using the given decoder (optimize the encoder as in the first property). If the distortion is small enough, terminate the algorithm.
- **Step 3** For each codeword **u** replace the corresponding code vector with the centroid of all input vectors that mapped to **u** in the encoder produced by step 1 (optimize the decoder as in the second property). Then repeat step 2.

The last detail to be addressed is the selection of the initial decoder. Clearly in an iterative technique such as this, a good initial selection can make a large difference in the number of iterations required for convergence to the final result. Several techniques have been employed (See for example [Ref. 7]). First, we can just select the appropriate number of input vectors from the image and use them as the code vectors in the initial codebook. Second, we can apply a scalar quantizer to each element of the vector and generate the number of values needed to form the code vectors. Lastly, we can use a technique known as splitting. In this technique we start with a codebook of size 1, which is just the centroid of the entire data set. Then we split the code vector by adding and subtracting a small vector from the original code vector and optimize this new codebook of size two. Then we split the two resulting code vectors into a codebook of size 4 and optimize this code book as well. We continue this pattern of splitting and optimization until the desired codebook size is reached. Of the initialization techniques described above, the splitting technique is most often used because it initializes each step with a good initial guess which limits the number of iterations required.

Now we present a two dimensional example to give an intuitive feel for how the splitting algorithm progresses. The data set for the example is presented in Figure 2.2. This data set has been chosen to have a simple structure in order to eliminate the need for many iterations at each splitting. In this example we attempt to generate a vector quantizer of size four. Step 1 is to form the codebook of size one by calculating the centroid of the entire data set (See Figure 2.3). Step 2 is to split the size one codebook into a size two codebook by adding and subtracting a small vector (See Figure 2.4). Step 3 is finding the vector subspaces which define the decision areas for the new codebook (See Figure 2.5). Step 4 is to calculate the centroid of each of the new vector subspaces and make these centroids the new code vectors (See Figure 2.6). Step 5 is to split the newly generated size two codebook into a size four codebook by adding and subtracting a small vector from each code vector (See Figure 2.7). Step 6 is to calculate new vector subspaces for each of the code vectors (See Figure 2.8). Step 7 is to calculate the centroid of each new subspace and assign them as code vectors (See Figure 2.9). Finally, step 8 is to find the vector subspaces corresponding to the decision regions for our final codebook (See Figure 2.10) and the algorithm is complete.

For this example it is easy to see where the code vectors for a vector quantizer of four should be placed; one at the centroid of each of the four clusters. This is the result produced by the LBG algorithm. However, we must keep in mind that this example was carefully contrived to eliminate the large number of iterations required for optimization at each splitting. A problem from a real data set, even if the dimension and size are the same as our example, would be much more computationally expensive.

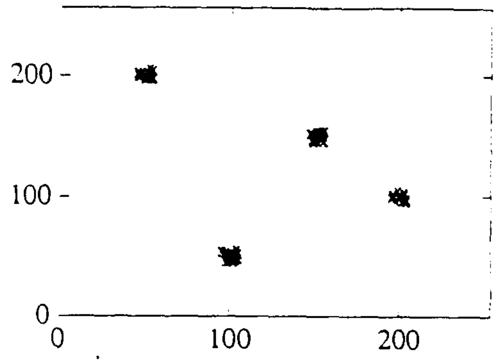
Figure 2.2 Original Data Set

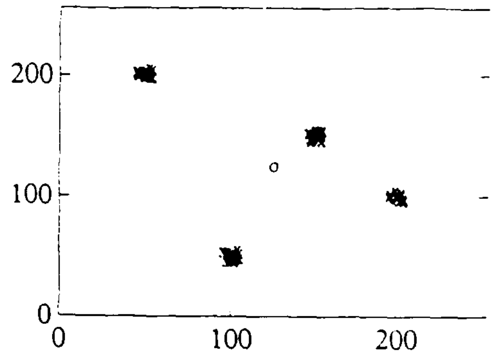
Figure 2.3 Step 1, Centroid of Data Set

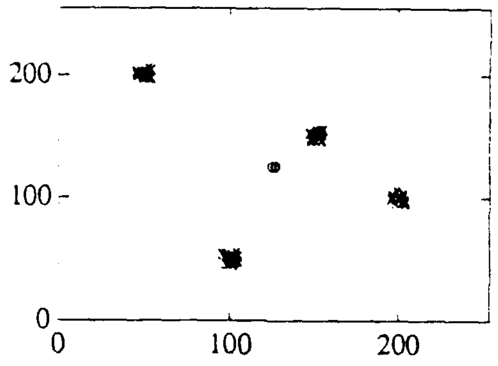
Figure 2.4 Step 2, First Point Splitting

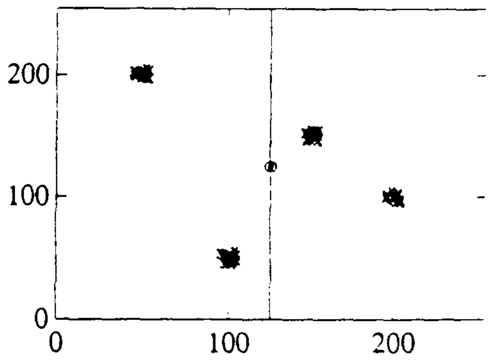
Figure 2.5 Step 3, New Subspaces

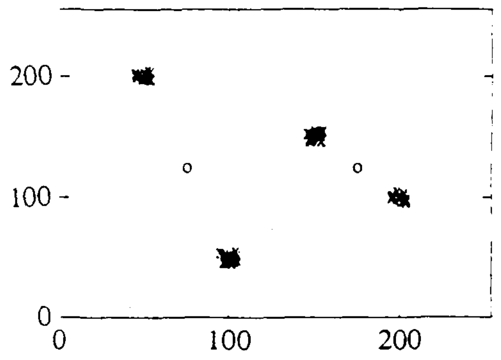

Figure 2.6 Step 4, New Centroids

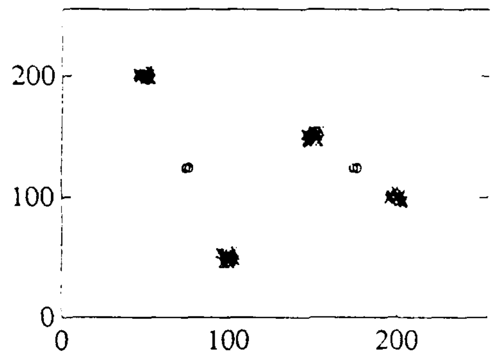

Figure 2.7 Step 5, Second Point Splitting

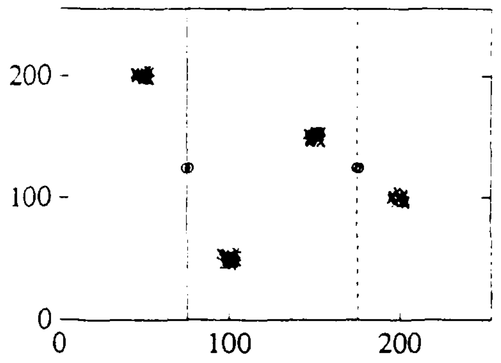

Figure 2.8 Step 6, New Subspaces

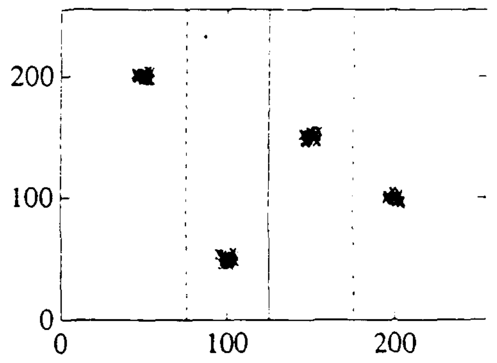

Figure 2.9 Step 7, New Centroids

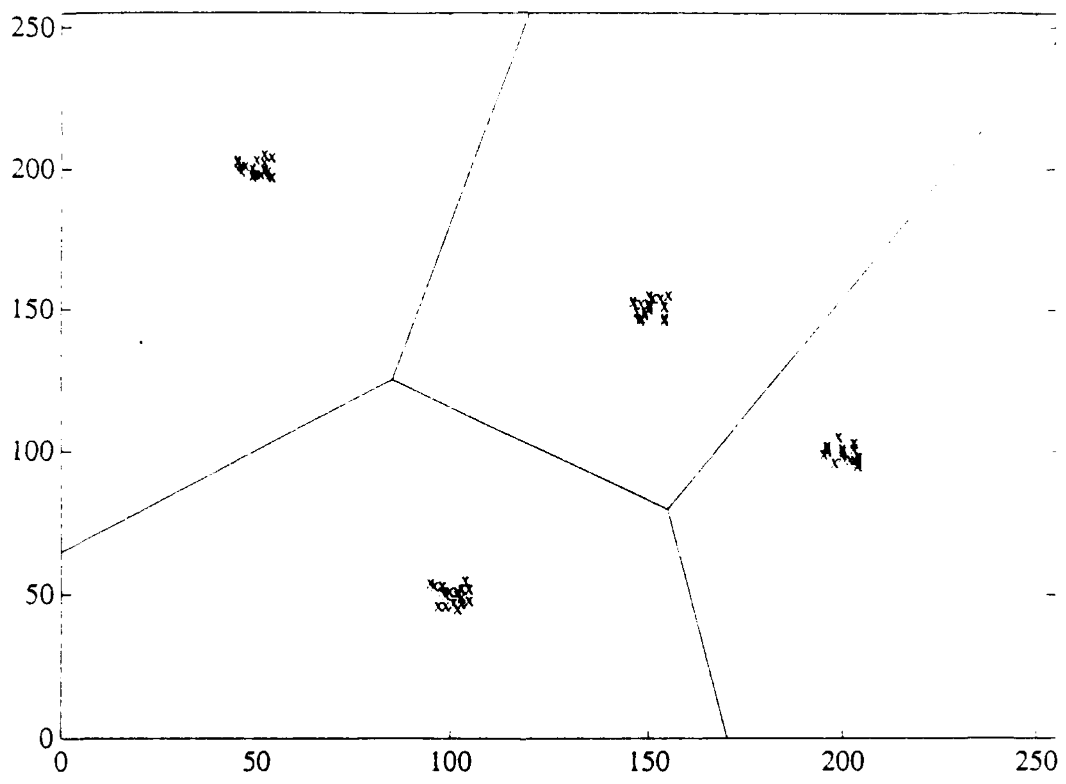

Figure 2.10 New Subspaces, Algorithm Complete

III. NEURAL NETWORKS

A. INTRODUCTION

Artificial Neural Networks [Ref. 8] have recently been the subject of intense research because of a desire to develop machines which can achieve human-like performance in such areas as speech and image recognition. After a lengthy period of inactivity in this area, the recent development of new algorithms, advances in analog VLSI techniques, and a new emphasis on parallel computing have contributed to major advances in this field.

Like their biological counterparts, neural networks rely on a large collection of simple but highly connected processing elements. This enables the neural network to avoid the sequential instruction processing characteristic of the von Neumann computer, and instead process many possible results in parallel. This property makes a neural network an attractive option to investigate in many recognition problems. Neural networks are also designed to adaptively update the interconnection weights between processing elements in an effort to improve their performance. This adaptive updating is termed "learning." This property allows a neural network to continue to function well despite changes in the statistics of the input data.

A neural network is a good tool in pattern recognition because of its ability to quickly categorize an input pattern in a previously learned category. However, there also exist different algorithms which are equally proficient at taking a data set and forming the occurring patterns into categories without supervision. That is, without external definition of the categories to be used by the neural network. Thus with some modifications, a neural network can be made to do a task which is very similar to vector quantization. If we can find the proper way to update the interconnection weights and the proper function for the processing elements, we should be able to find a configuration that is capable of duplicating the results of the LBG algorithm which we saw in the previous chapter. In the following sections we will briefly discuss the difference between unsupervised and supervised learning, and how neural networks can be applied to the problem of vector quantization.

B. NEURAL NETWORK LEARNING

Each processing element of the neural network is connected to many inputs $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$ (See Figure 3.1). These inputs could originate directly from the input to the network, or some or all could arrive from the output of another processing element. Each input to the processing element has an associated weight w_i, which describes the strength of the connection between the associated input node and this processing element. Each processing element has an activation level which is a function of the inputs and weights. One of the most common activation formulas is

$$y = f(\sum_{i=0}^{N-1} w_i x_i - \theta) \tag{3.1}$$

where θ is some threshold. This is just a weighted sum which is thresholded and subjected to a function f, which is usually nonlinear.

Typical neural networks are made up of many of these processing elements which are arranged and interconnected in some pattern. This pattern, the activation formula discussed above, and the scheme for adaptively updating the weights for each processing element are the items which characterize each type of neural network.

A final property which characterizes a neural network is the manner in which it is trained. There are two main categories, namely supervised and unsupervised.

1. SUPERVISED LEARNING

In supervised learning, the neural net is provided a set of desired output values for each set of input values presented to the network. These desired output values are used in order to update the interconnection weights. A good example of a neural network which uses supervised learning is the backpropagation network [Ref. 9], which is arranged as in Figure 3.2. The activation function for a typical implementation of backpropagation algorithm is

$$f(\alpha) = \frac{1}{1 + e^{-(\alpha-\theta)}} \tag{3.2}$$

which is known as a sigmoid logistic function. The training of the network proceeds as follows:

- **Step 1** Initialize the interconnection weights to small random values.
- **Step 2** Present a set of input values and corresponding desired output values to the network.
- **Step 3** Apply the activation formula for each processing element until the output values have been calculated.
- **Step 4** Update the interconnection weights starting with the output layer and moving downwards using the formula

$$w_{ij}(t+1) = w_{ij}(t) + \eta\delta_j x_i \tag{3.3}$$

where w_{ij} is the interconnection between node i of the previous layer and node j of the current layer, x_i is the activation level of node i , η is the learning rate. The backpropagated error is

$$\delta_j = \begin{cases} y_j(1-y_j)(d_j - y_j) & \text{for an output node} \\ x_j(1-x_j)\sum_k \delta_k w_{jk} & \text{for an intermediate node} \end{cases} \qquad (3.4)$$

where y_j is the activation level of node j on the current layer, and d_j is the desired output for node j.

Steps 2-4 are repeated until the network weights have converged or the error between the output and desired signals is sufficiently low. This method works well for a case such as speech recognition where we can collect a large quantity of sample data with the correct classification appended to allow training of the network. However a network of this sort is of little use for a problem like vector quantization in which the neural network must form the desired categories without any external guidance.

2. UNSUPERVISED LEARNING

A good example of a neural network algorithm that utilizes unsupervised learning is the competitive learning network shown in Figure 3.3. This algorithm is designed to take the set of input vectors and use them to form a set of categories; one category for each node on the second level of the network. This is accomplished by measuring the proximity of each input vector to the set of weights for each node on the second level and adaptively adjusting the weights of the closest node towards the input vector. After sufficient training, the network should categorize all input vectors which are similar into the same category based on their distance from the weight vector of each node.

The training of the competitive learning algorithm proceeds as follows.

- **Step 1** Initialize the weights from the N input nodes to the M output nodes with small random numbers.

- **Step 2** Present an input vector from the data set.

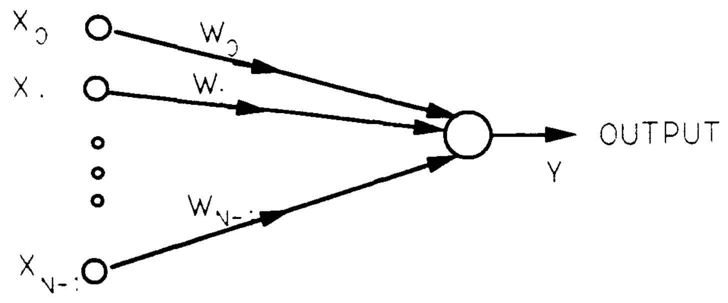

Figure 3.1 Neural Network Processing Element

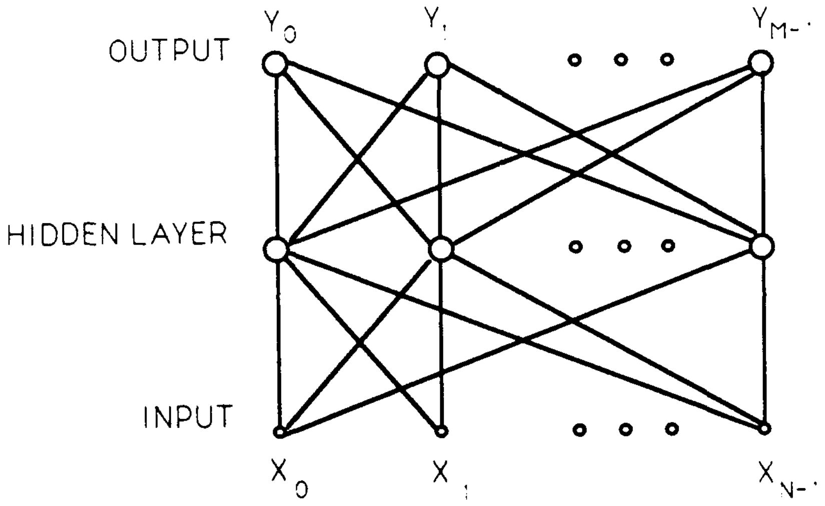

Figure 3.2 Backpropagation Network

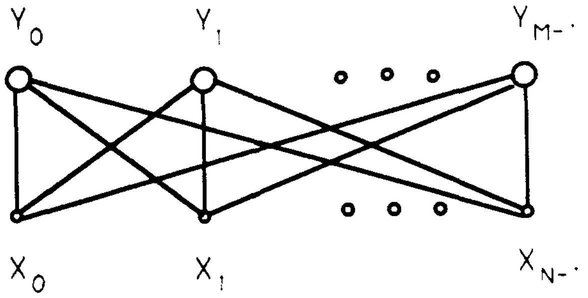

Figure 3.3 Competitive Learning Network

- **Step 3** Compute the distance d_j, between the input vector and the weights of each output node j using the formula

$$d_j = \sum_{i=0}^{N-1} [x_i(t) - w_{ij}(t)]^2 \tag{3.5}$$

where $x_i(t)$ is the input to node i at time t, and $w_{ij}(t)$ is the interconnection weight from input node i to output node j at time t. Note that this distance measure is just the unnormalized MSE between the input vector and the weight vector of output node j.

- **Step 4** Select the output node j^* which is closest to the input vector.

- **Step 5** Update the weights of the closest output node j^* using the expression

$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)(x_i(t) - w_{ij}(t)) \tag{3.6}$$

where $\eta(t)$ is the time dependent learning rate.

- **Step 6** Get the next input vector and return to step 2.

We continue training the network until convergence is obtained or the average error for the entire data set is less than some threshold value.

It is not hard to see the resemblance between vector quantization and the task performed by the competitive learning algorithm. To implement VQ, we just present each block of the image as an input vector and train the neural network until it converges. Then the weight vectors produced for each output node are the code vectors for the VQ codebook, and the indices of the output nodes are the corresponding codewords. After training, the weights are fixed and the codebook is transmitted to the receiving site. Then each block to be transmitted is submitted to the neural network. The index of the closest output node to the input is transmitted as the VQ codeword. At the receiving site, the codeword is used as the argument in a lookup table in which the codewords and the corresponding code vectors are stored. The code vector chosen is then converted into an image block which serves as an approximation to the original block. After the codewords for all the blocks in the image are transmitted and decoded, the final reproduction image is assembled from the code vector approximations.

The competitive learning algorithm is now applied to the two dimensional VQ example presented in the previous chapter. The trajectories of the code vectors are presented in Figure 3.4. Notice that the algorithm attempts to represent the data with a single code vector. This occurs because the code vector that is closest for the first input vector continues to be the closest for all subsequent input vectors. Thus none of the other code vectors are ever utilized and their weights are never updated. An algorithm such as this clearly does not utilize all its code vectors and thus cannot produce an optimum vector quantizer. In the next section we examine modifications to the competitive learning algorithm which improve its performance as a vector quantizer.

C. FREQUENCY SENSITIVE COMPETITIVE LEARNING

As shown in the previous section, the principal problem with using competitive learning as a vector quantizer is the under-utilization of the output nodes. This problem has been addressed in the literature and several possible solutions have been presented. In [Ref. 10], an algorithm referred to as the Self Organizing Map (SOM) is introduced. In the SOM, a neighborhood is defined about the closest output node and this neighborhood is used to update more than one output node at a time. In this technique the update formula becomes

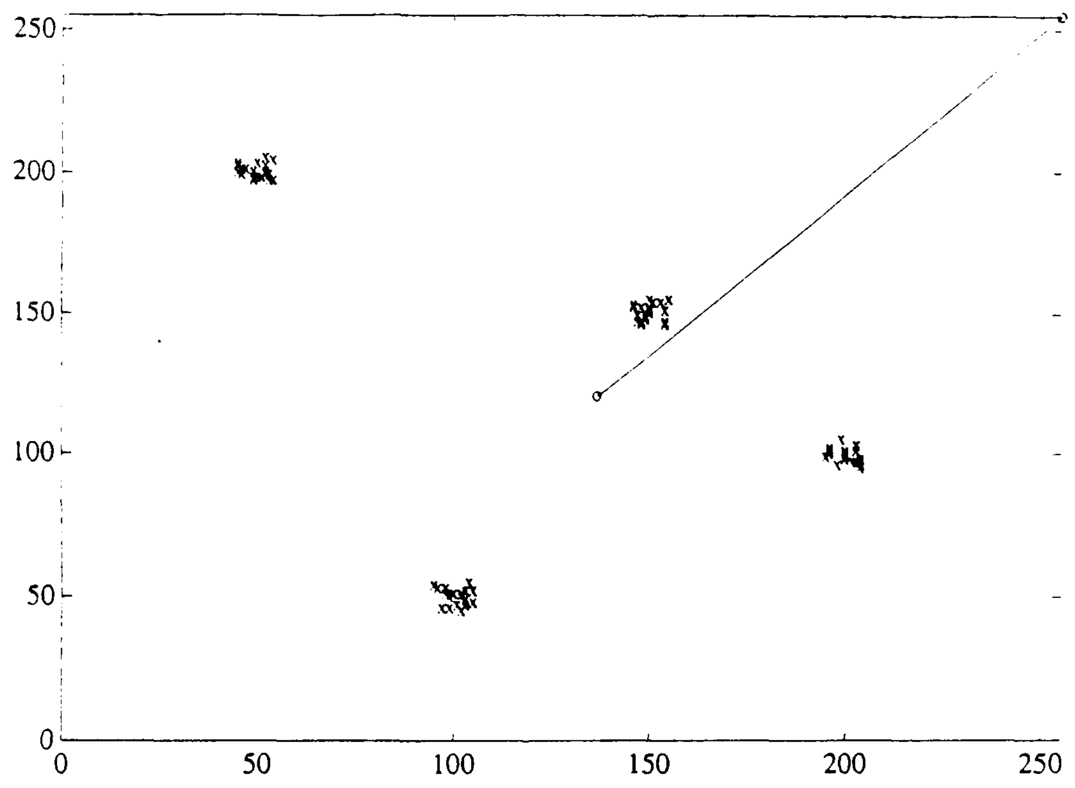

Figure 3.4 Competitive Learning 2-D Example

$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)[x_j(t) - w_{ij}(t)],\ j \in N(j^*, t) \qquad (3.7)$$

where j^* is the index of the closest output node and N is the neighborhood defined about the closest output node. This neighborhood is started with a large size to encourage the updating of many output nodes, and then gradually shrunk with time as the network converges to generate more fine structure. Finally, the neighborhood shrinks to a single node which allows each node to be updated independently. At this point the SOM algorithm is identical to the original competitive learning. We can see that the improvement in output node utilization comes from establishing a good distribution of weight vectors throughout the input vector space and then allowing the network to converge. The drawback of this technique is that the resulting network takes an excessive number of iterations to reach convergence.

Another technique termed *adding a conscience to competitive learning* is presented in [Ref. 11]. In this algorithm we generate a new variable, p_j, for each output node which represents the percentage of the time that a particular node is the closest to the input vector. This variable is initialized to zero and updated by :

$$p_j(t+1) = \begin{cases} (1-B)p_j(t) + B & \text{for } j = j^* \\ (1-B)p_j(t) & \text{for } j \neq j^* \end{cases} \qquad (3.8)$$

where B is a constant which is chosen small enough to prevent random fluctuation in the input data from having too large an effect on p_j. Then a bias term, b_j, is calculated using $b_j = C(1/M - p_j)$, where C is termed the bias constant. This bias term is then applied to the distance measure for each output node, and the closest node is chosen based on this biased distance, $d_j - b_j$. The result of these modifications is to penalize the output nodes that have won the competition frequently. This produces a very uniform output node utilization. This algorithm has the advantage of converging quickly while maintaining good output node utilization, but requires twice as many distance calculations as the original competitive learning algorithm.

A variation on the *conscience* technique discussed above is Frequency Sensitive Competitive Learning (FSCL) [Ref. 12]. In this algorithm the distance d_i, between the input vector and the output node weight vector is modified by:

$$d_i^* = d_i \; g(u_i) \tag{3.9}$$

where u_i is the number of times the output node i has won the competition and g is termed the fairness function, with $g(u_i) = u_i$ in most cases. The effect of this modification is to increase the modified distance for those nodes which win frequently. Over many training iterations, the result is a remarkably even node utilization. This algorithm preserves the fast convergence of the *conscience* method and also requires us to update only one set of weights for each input vector. In addition, the algorithm requires only one set of distance calculations and is thus much faster than the *conscience* method. The FSCL vector quantizer is the basic building block which will be used in the algorithms in the next chapter.

We first apply the FSCL vector quantizer to the same 2-D example for which the competitive learning algorithm failed. The trajectories of the code vectors are shown in Figure 3.5. The FSCL clearly solves the problem of node utilization and produces the same result as the LBG algorithm.

The FSCL has been applied to the vector quantization of images [Ref. 13] and some interesting results have emerged. Figure 3.6 shows the number of training iterations required by the FSCL and LBG algorithms. For a small codebook, the FSCL has a sizable computational advantage, while for larger codebooks the LBG algorithm is more efficient. To get an idea of how codebook size affects reproduction quality, we have applied the FSCL algorithm using various codebook sizes. The

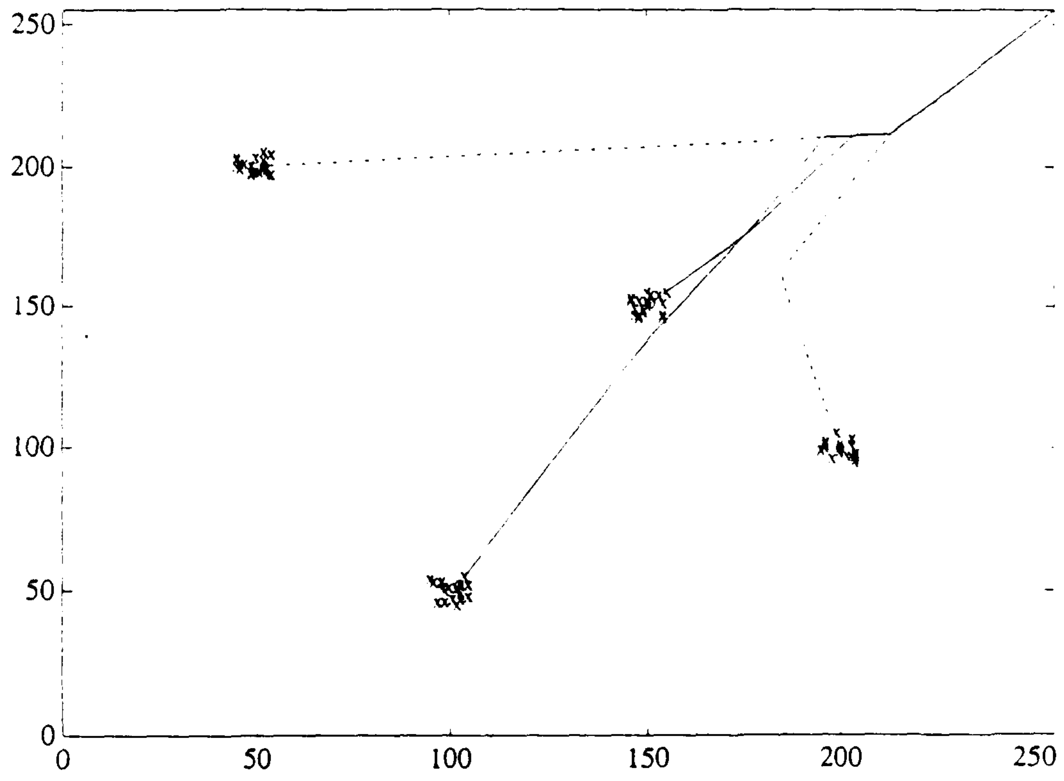

Figure 3.5 FSCL 2-D Example

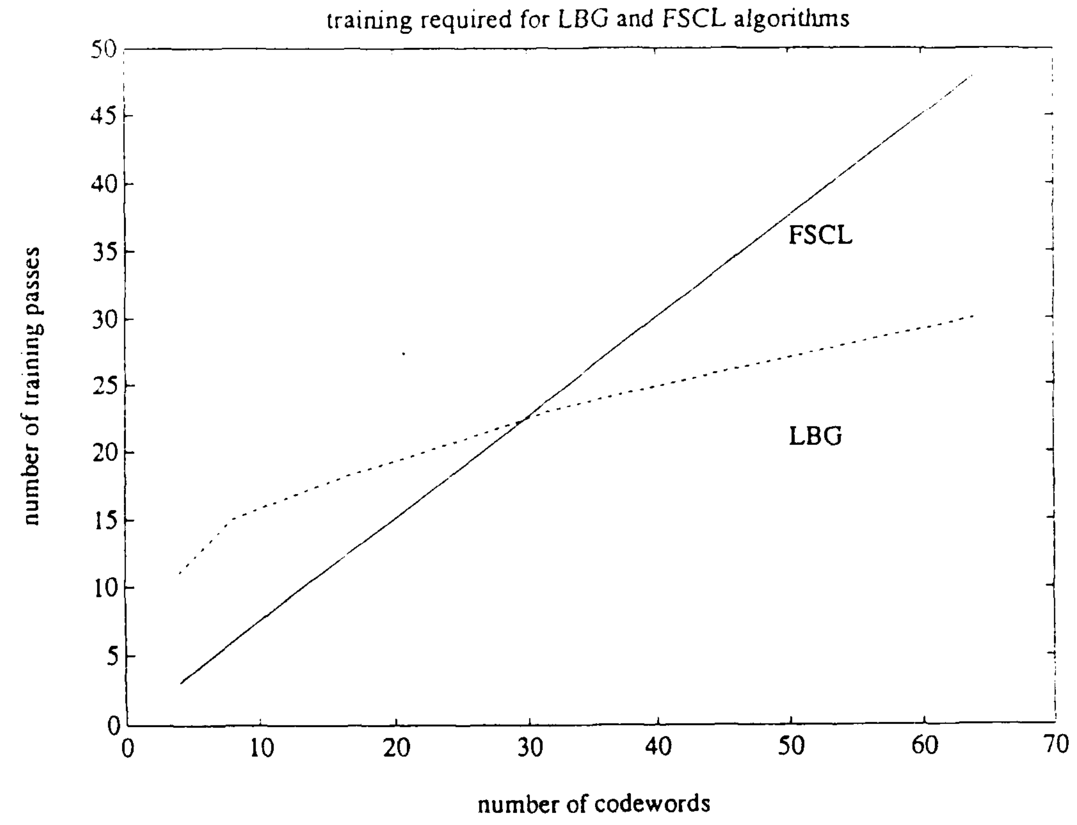

Figure 3.6 Training Required For LBG and FSCL Algorithms

Figure 3.7 Original Image

Figure 3.8 FSCL Using a Size 16 Codebook and a 2x2 Block

Figure 3.9 FSCL Using a Size 64 Codebook and a 3x2 Block

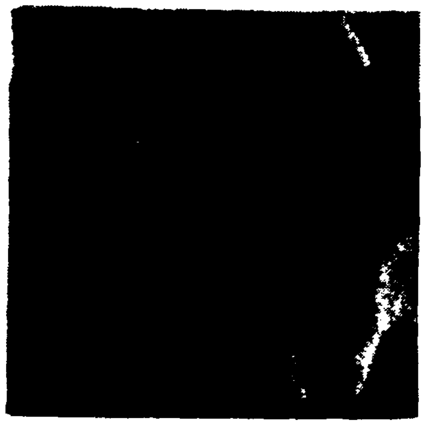
Figure 3.10 FSCL Using a Size 512 Codebook and a 3x3 Block

original image is 256×256 pixels (See Figure 3.7) and was divided into blocks of various sizes to produce a data rate of 1 bit/pixel for each example. Figure 3.8 shows an image produced with a block size of 2×2 and a codebook size of 16. Figure 3.9 shows an image produced using a block size of 3×2 and a codebook size of 64. Figure 3.10 shows an image produced using a 3×3 block and a codebook size of 512. We can clearly see that the larger codebooks produce a much better quality of reproduction at the same data rate. This leaves us with the question of how to get the good reproduction quality of large codebooks while also taking advantage of the computational efficiency of the FSCL algorithm for generating small codebooks. The next chapter demonstrates several techniques that can be applied to the FSCL algorithm which allow us to form large codebooks without the excessive amount of training required by the orig[illegible]al algorithm.

IV. ALGORITHM DEVELOPMENT

A. INTRODUCTION

The previous two chapters provided an overview of vector quantization, and how neural networks have been applied to this problem. Here we investigate the limitations of existing algorithms, and propose modifications which substantially reduce the computational requirements without significant loss in performance.

As we saw in Figures 3.8-3.10, the reproduction quality of a vector quantizer depends strongly on the dimensionality of the vector utilized. We wish to use the maximum dimensionality possible, but we are limited by the fact that the codebook size grows exponentially with increasing vector dimension. Whether we plan to implement the neural network by simulation or in hardware, this limitation introduces significant difficulties.

In the case of simulation, the large capacity memory chips available today allow us to implement very large codebooks. However, we can see from Figure 3.6 that for very large codebooks, the neural network algorithm has a substantially higher computational cost than the Linde, Buzo, and Gray (LBG) algorithm. So in order to make the neural network simulation useful, we must limit ourselves to small codebooks and thus poor performance, or find a way to form a codebook with a large effective size by combining many smaller codebooks.

In the case of hardware implementation, the computational disadvantage of the neural network for large codebook size is substantially mitigated by the advantage gained from parallel processing. However in this case, the codebook size is now limited by the number of processing elements which can be implemented in hardware. Even with expected advances in neural network hardware, it is still important to maximize the effective codebook size for a given number of processing elements.

In the following sections we investigate algorithms which improve the performance for both hardware and simulation implementations. These algorithms allow a large codebook to be formed from many small codebooks, and allow a large effective code book to be formed using a substantially smaller number of processing elements.

The vector quantizers we have examined so far are optimal in two senses. First the codebook formed produces the minimum MSE possible for the training data utilized, and secondly the encoder always picks the codeword corresponding to the vector which produces the least distortion for any given input vector. This type of algorithm is called full search vector quantization (FSVQ), and it must calculate a number of distortions equal to the size of the codebook for each vector processed. As noted above, this property makes full search codes impractical except for the case of small codebooks.

We now consider algorithms that produce codes which are suboptimal in both senses mentioned above. They may not produce a codebook which produces the minimum MSE for the training data, and they may not select the codeword corresponding to the smallest distortion available. However these algorithms produce codebooks which have structure that dramatically reduces the computational effort required for a given codebook size. Although the performance is degraded relative to a full search algorithm, the suboptimal algorithm can offer such a large reduction in complexity that a larger codebook may be implemented. This in turn can provide better performance at a smaller computational cost than the full search algorithm. These algorithms are described below.

B. TREE SEARCHED VECTOR QUANTIZATION

The TSVQ [Ref. 14] design was developed in an attempt to reduce the number of distance calculations which must be made to encode a vector. In the neural network implementation, not only does the software simulation option also benefit from this reduction in distance calculations, but we also see a reduction in the amount of training required. This improvement stems from the fact that the structure of the TSVQ produces data subsets for which the basic FSCL algorithm vector quantizer converges more quickly.

The TSVQ algorithm is a structure which causes us to search a sequence of smaller codebooks rather than a single large one. This is accomplished by arranging many small vector quantizers in a tree structure as shown in Figure 4.1. The tree is searched starting with the root, and each search of the smaller vector quantizers advances one level through the tree. An m level TSVQ is characterized by the m-tuple $\mathbf{R} = (R_1, R_2, \ldots R_m)$, which describes the number of bits encoded at each level of the tree. So each vector quantizer at level j would have 2^{R_j} codewords and $2^{\sum_{i=1}^{j-1} R_i}$ vector quantizers are required to complete level j. The codebook size for the entire structure is $2^{\sum_{i=1}^{m} R_i}$.

The encoding of a vector proceeds by first applying the input vector, $\mathbf{x}$, to the vector quantizer at the root of the tree structure. This produces the closest code vector, $\mathbf{y}_1$, which is our first estimate of $\mathbf{x}$, and the first R_1 bits of the channel codeword. This R_1-tuple, $\mathbf{u}_1$, also serves as the index of the vector quantizer to be searched in the next level. Thus each codeword in level one provides a mapping to a vector quantizer in level two. We then present $\mathbf{x}$ to the 2^{R_2} size vector quantizer selected at level two which produces a new estimate $\mathbf{y}_2$ and the second portion of the channel codeword $\mathbf{u}_2$. We use the vector $(\mathbf{u}_1, \mathbf{u}_2)$ to choose the vector quantizer to search at the third level. This process continues until the final level is reached. At this point, we have produced our final estimate $\mathbf{y}_m$ and the complete channel codeword, $\mathbf{u} = (\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_m)$. This structure allows us to encode a vector using only $\sum_{i=1}^{m} 2^{R_i}$ distance calculations in contrast with the 2^R calculations required for the full search method ($R = R_1 + R_2 + \ldots R_m$) . Table 4.1 shows some examples of how large the computational savings for the encoding step can be for TSVQ. The **R** vector listed in the table describes the architecture of the particular TSVQ structure used in the example. This notation is explained later in this section.

TABLE 4.1: Number of Encoding Distance Calculations Required

			Distance Calculations	
R	Block Size	Codebook Size	FSVQ	TSVQ
(2,2)	2×2	16	16	8
(3,2)	3×2	64	64	16
(3,3,3)	3×3	512	512	24

The training of the TSVQ proceeds one level at a time. We first apply the entire training set to the FSCL vector quantizer at the root of the tree until convergence is obtained. We then use the codebook produced to divide up the data set into R_1 subsets based on their proximity to the newly generated code vectors. The new subsets are then applied to the R_1 vector quantizers on level two. We proceed in this way until the vector quantizers at the final level, m, have been trained. Each vector quantizer in the tree is initialized by randomly selecting vectors from the appropriate training set. This type of initialization speeds convergence of the neural networks.

This structure allows us to greatly reduce the number of distance calculations necessary for the software simulation case. This is true because the path through the tree allows us to ignore the vast majority of code vectors which are far from the input vector. TSVQ also displays a property which is termed *graceful degradation*. This means that if the codeword must be truncated due to channel capacity considerations, it will still be possible send a good estimate of the data for this new lower data rate. This is in contrast to the full search method, whose codeword conveys no useful information if it is truncated. An added benefit of this method is that the structure imposed on each of the data subsets applied to the FSCL vector quantizers causes them to converge more quickly. This provides a substantial reduction in training required for both software and hardware implementations.

A final benefit of the method is the large reduction in the number of processing elements required for hardware implementation. Since the TSVQ algorithm updates only the weights of the vector quantizers of the path taken for each input vector applied, these are the only vector quantizers that must be realized in hardware. Thus we can convert the hardware implementation from a tree structure to a linear structure (see Figure 4.2) along with memory and a system to load the appropriate weights for each level. Thus we can reduce the number of processing elements required from $\sum_{i=1}^{m} \prod_{j=1}^{i} R_j$ to $\sum_{i=1}^{m} R_i$. Table 4.2 shows some examples of the number of processing elements required if the TSVQ is implemented in hardware using tree structure and linear structure. For larger block sizes and code book sizes, the savings is substantial.

TABLE 4.2: Number of Processing Elements Required

			PE's Required	
R	Block Size	Codebook Size	Tree Structure	Linear Structure
(2,2)	2×2	16	20	8
(3,3)	3×2	64	72	16
(3,3,3)	3×3	512	574	24

For the simulations, a single 256×256 pixel image was utilized. This image was divided into blocks of various sizes to achieve a data rate of 1 bit/pixel for each example. The 1 bit/pixel provided a standard to allow comparisons between examples with different codebook sizes, and provided a challenging enough problem to allow

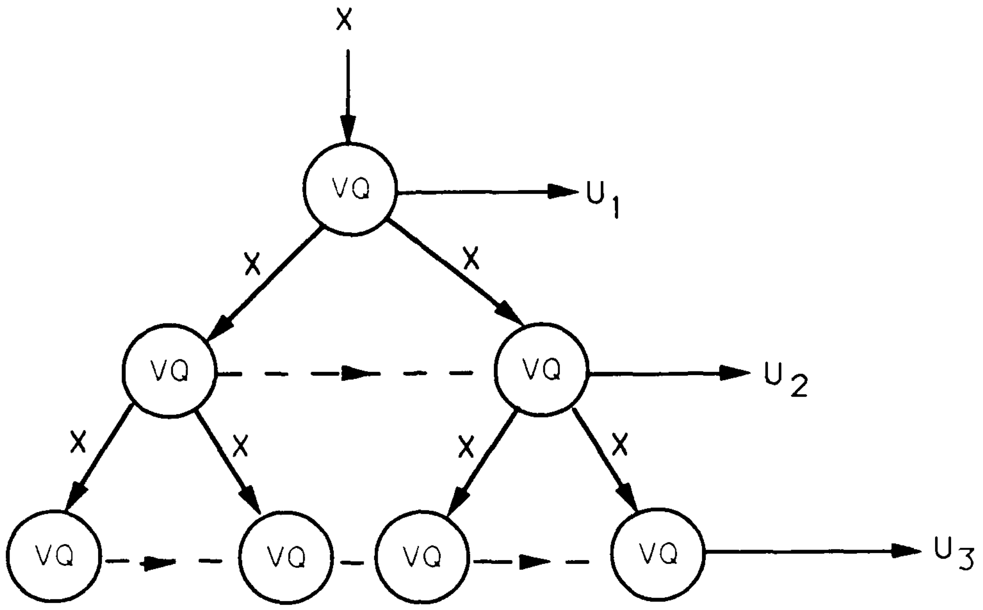

Figure 4.1 Tree Search Vector Quantization

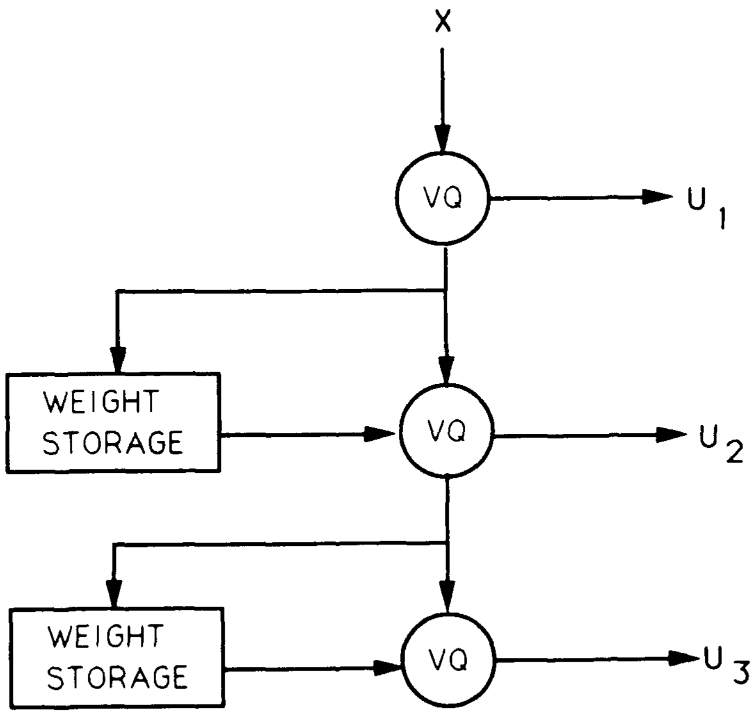

Figure 4.2 Linear Hardware Implementation Of TSVQ

good comparisons to be made.

The simulation results for TSVQ are shown in Figures 4.3-4.6. The original image is shown in Figure 4.3. TSVQ with a block size of 2×2, and a size 16 codebook constructed from five size 4 codebooks arranged in a two level tree, is shown in Figure 4.4. TSVQ with a block size of 3×2, and a size 64 codebook constructed from nine size 8 codebooks arranged in a two level tree, is shown in Figure 4.5. TSVQ with a block size of 3×3, and a size 512 codebook constructed from 73 size 8 codebooks arranged in a three level tree, is shown in Figure 4.6. It is easy to see the strong effect of codebook size on performance by noting the improvement in subjective quality as the codebook size is increased from 16 to 64 to 512. In particular, the larger codebook sizes display an image that appears sharper because the small code books does not contain a sufficient number of code vectors to represent edges well. Also, the small code book does not contain code vectors with enough different grey scales to reproduce gradually changing intensities, such as those in the top of the hat or near the beam to the left of the hat. This is confirmed by the MSE performance, which is displayed in Figure 4.7. Comparing the MSE performance of TSVQ to the full search algorithm, we can see that the loss of performance is very small. This is reinforced by comparing Figures 4.4-4.6 for TSVQ and Figures 3.8-3.10 for full search, which show that the degradation caused by use of the TSVQ method is small in the subjective sense as well.

To give an idea of the refinement taking place at each level, each stage of the three stage TSVQ example in Figure 4.6 is shown in Figures 4.8-4.10. The improvement taking place at each level is clear. We can also get a good idea of what would be reconstructed if the code were truncated. Figure 4.8 corresponds to 0.33 bits/pixel, Figure 4.9 corresponds to 0.67 bits/pixel, and Figure 4.10 corresponds to 1.0 bit/pixel. It is apparent that a degraded but nevertheless useful image is still available if the

Figure 4.3 Original

Figure 4.4 TSVQ Using a Size 16 Codebook and a 2x2 Block

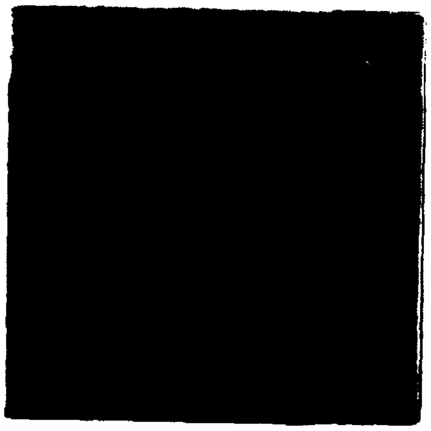

Figure 4.5 TSVQ Using a Size 64 Codebook and a 3x2 Block

Figure 4.6 TSVQ Using a Size 512 Codebook and a 3x3 Block

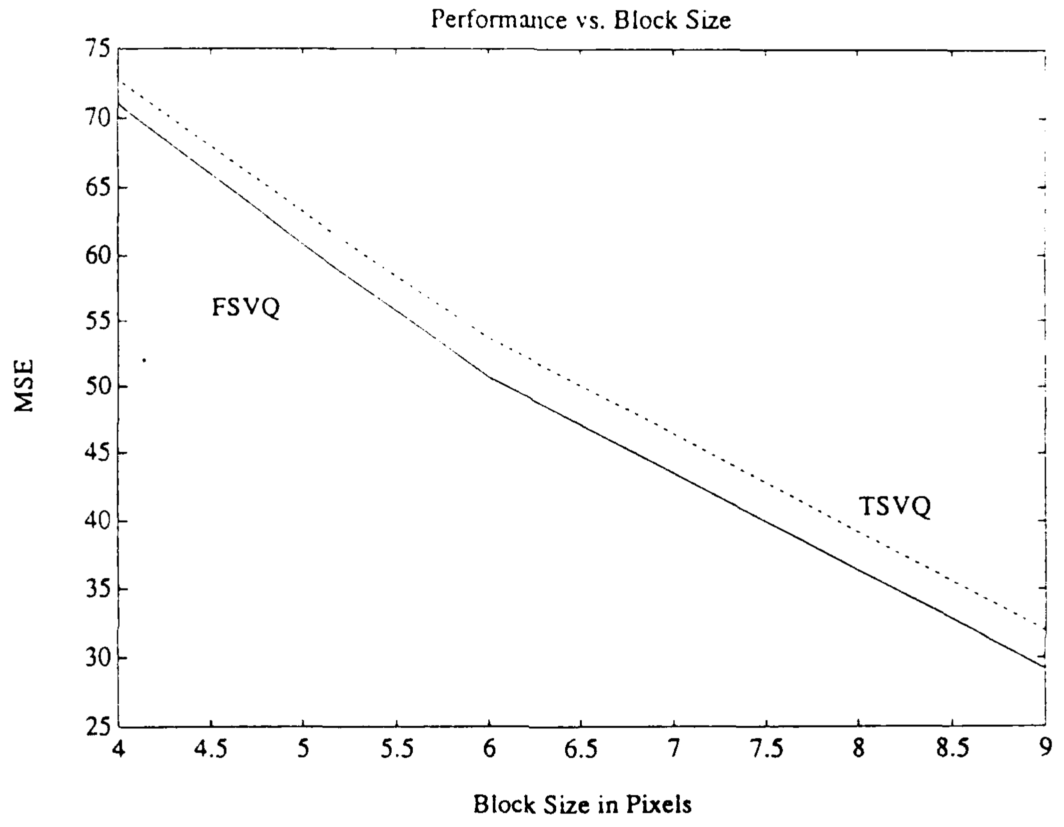

Figure 4.7 Performance vs. Block Size

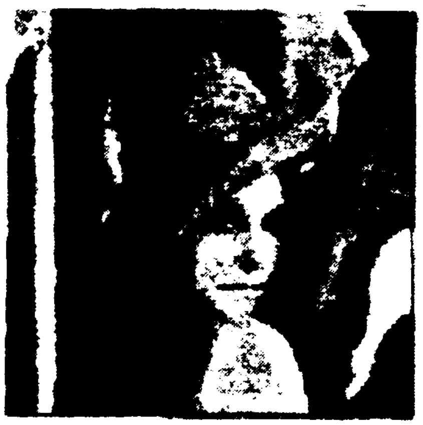

Figure 4.8 TSVQ Using 3x3 Block
First Stage

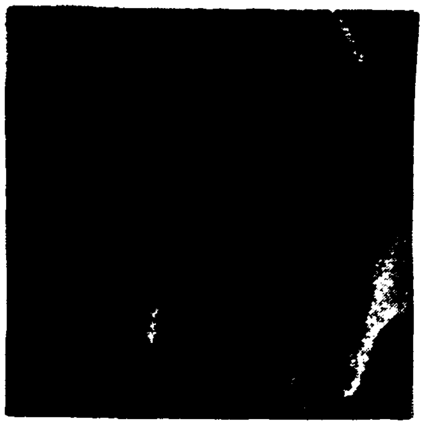

Figure 4.9 TSVQ Using 3x3 Block
Second Stage

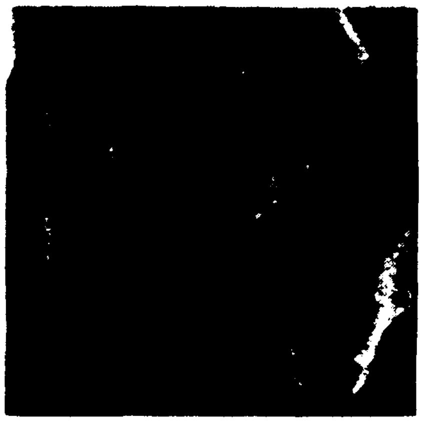

Figure 4.10 TSVQ Using 3x3 Block
Third Stage

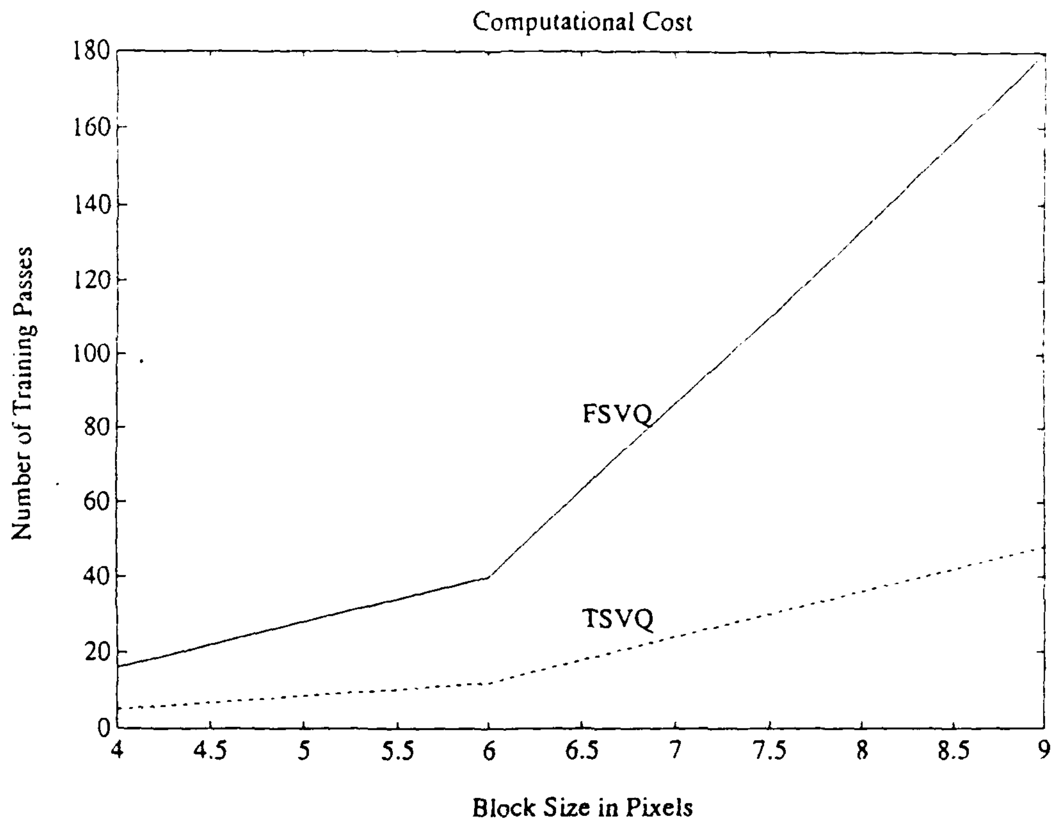

Figure 4.11 Computational Cost

code is truncated. This is the property termed previously as *graceful degradation.*

The improvement in computational cost can be seen in Figure 4.11. For the three examples, the savings varied from 68 to 72 percent. In other words the TSVQ algorithm required only about 1/3 to 1/4 the computation. As stated before, this advantage is a result of utilizing smaller more efficient codebooks to form a single large effective codebook. This large computational advantage is gained at an very modest loss of performance. This makes the TSVQ an extremely attractive alternative to the FSCL algorithm.

C. MULTI STAGE VECTOR QUANTIZATION

We saw in the last section that the TSVQ algorithm offers many advantages for neural network vector quantizers, but that some troublesome limitations remain. First, for both TSVQ and FSVQ, the load on the channel of transmitting updates for very large codebooks can be excessive. Second, even though TSVQ can reduce the training effort, a large number of passes through the image is still required for good performance. Finally, although TSVQ produces a codebook with structure, it actually increases the storage required for the code book. We now examine the application of a technique termed Multiple Stage Vector Quantization (MSVQ) [Ref. 15] to the basic FSCL vector quantizer. This technique has the advantage of further reducing the computational cost and allowing a very efficient hardware implementation.

Like TSVQ, MSVQ has two or more levels, but instead of working with the original input vector at each stage as in TSVQ, MSVQ attempts to encode the error generated at the previous level. An m level MSVQ (see Figure 4.12) can be described by the m-tuple $\mathbf{R} = (R_1, R_2, \ldots, R_m)$, where R_i is the number of bits used to encode the error at level i of the MSVQ. The first level of the MSVQ is just a normal FSCL vector quantizer. The input vector, $\mathbf{x}$, is applied to the vector quantizer at level one

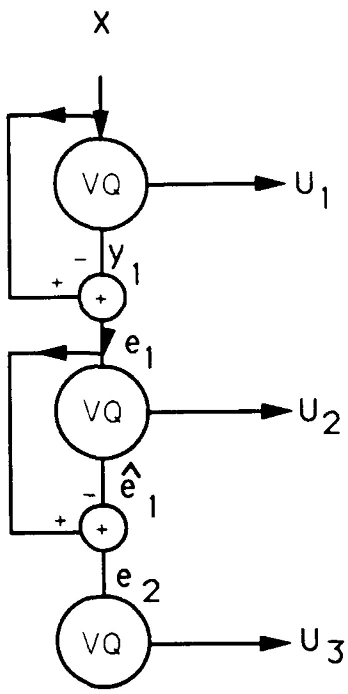

Figure 4.12 Multi Stage Vector Quantization

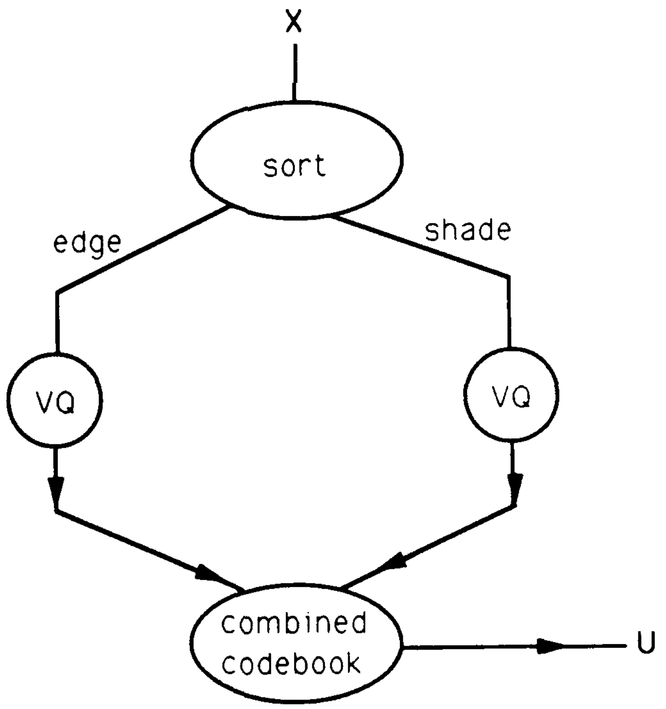

Figure 4.13 Classification Vector Quantization

and the first estimate, $\mathbf{y}_1$, is produced along with the first R_1 bits of the channel codeword, $\mathbf{u}_1$. Next, the first error vector is formed by taking the vector difference, $\mathbf{x} - \mathbf{y}_1$. This error vector, $\mathbf{e}_1$ is then applied to the size 2^{R_2} vector quantizer at level two, which produces an estimate of the error vector, $\hat{\mathbf{e}}_1$, and the next R_2 bits of the channel codeword. So at the second level, our estimate of the input vector is the vector sum $\mathbf{y}_2 = \mathbf{y}_1 + \hat{\mathbf{e}}_1$. In the following stages, we continue to form an error vector from the previous stage and use a FSCL vector quantizer to encode this error. Each stage produces an estimate for the error and a portion of the channel codeword. At the last stage, the error vector $\hat{\mathbf{e}}_{m-1}$ is encoded and the final estimate of the input vector is available by performing $\mathbf{y}_m = \mathbf{y}_1 + \hat{\mathbf{e}}_1 + \hat{\mathbf{e}}_2 + \ldots + \hat{\mathbf{e}}_{m-1}$, and the full channel codeword $\mathbf{u} = (\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_m)$.

For encoding, MSVQ requires $\sum_{i=1}^{m} 2^{R_i}$ distance calculations which is the same as TSVQ and much less than the 2^R required for FSVQ. However, the MSVQ requires only m vector quantizers and thus m small codebooks to be stored as compared with $\sum_{i=1}^{m-1} \Pi_{j=1}^{i} R_j$ for TSVQ. Table 4.3 shows the difference in the number of codebooks required by TSVQ and for some of the examples used in simulations. This reduces the total number of code vectors to be stored from $\sum_{i=1}^{m} \Pi_{j=1}^{i} R_j$ for TSVQ and 2^R for FSVQ to $\sum_{i=1}^{m} 2^{R_i}$ for MSVQ. Table 4.4 shows the total number of code vectors which must be stored for several examples of FSVQ, TSVQ, and MSVQ. We can see that there is a storage price to be paid for the computational advantage of TSVQ, but the MSVQ provides a large reduction in both. This dramatically reduces both storage requirements and the load on the channel from transmitting codebook updates. Table 4.5 shows the extra load on the channel for each of the three algorithms assuming that the codebook is updated with each frame. The advantage of MSVQ in this regard for large codebooks is apparent.

As with TSVQ, the training of the MSVQ proceeds one level at a time. We apply the original data set to FSCL vector quantizer at the first level until convergence is obtained. Then we pass the data through the trained vector quantizer and compute the error vector between each input vector and the closest code vector. This forms a new data set which is a collection of the first stage errors. This first stage error data set is then applied to the vector quantizer on the second level until convergence is obtained: then it is applied a final time to compute the second stage error vectors. This continues until the last stage has been trained. Although the data subsets produced by MSVQ do not have the same desirable structure as the data subsets from TSVQ, there are far fewer codebooks for MSVQ to train. Indeed we find that the smaller number of codebooks outweigh the larger convergence time in all cases except for very small overall codebooks. Thus the MSVQ requires significantly fewer training passes than TSVQ to reach convergence.

It is useful at this point to examine the differences between MSVQ and TSVQ. Both methods produce a multi-level process, but the processing at each level is significantly different. The TSVQ algorithm presents the original data vector at each level, while the MSVQ presents the residual error at each level. TSVQ has an ever increasing number of vector quantizers at each level, while MSVQ has a single vector quantizer at each level. TSVQ provides increasingly accurate estimates of the input at each level by systematically dividing the higher dimensional vector space into smaller and smaller subspaces into which the input must fall. MSVQ provides an initial estimate at the first level, and provides a better estimate at each level by continuing to add smaller and smaller correction terms in a way similar to the method of successive approximations. Each of these corrections is a result of performing vector quantization on the error subspace of the preceding level. In TSVQ, the code vectors at intermediate levels are not actually utilized for reconstruction; they are only used as pointers to direct the algorithm to the appropriate vector quantizer at the final level. Only code vectors of the vector quantizers at the final level are actually used in the image reconstruction.

The reconstructed images for MSVQ are presented in Figures 4.14-4.17. As in the previous results, the simulations were conducted on a single test image of 256×256 pixels. This image was divided into blocks of various sizes chosen to yield a data rate of 1 bit/pixel for each reconstruction using a variety of codebook sizes. The original image is presented in Figure 4.3. MSVQ using a 2×2 block and a codebook size of 16 is presented in Figure 4.14. This codebook was generated using a two level architecture containing two code books of size 4. MSVQ using a 3×2 block and a codebook of size 64 is presented in Figure 4.15. This codebook was generated using a two level architecture containing two codebooks of size 8. MSVQ using a 3×3 block and a code size of 512 is presented in Figure 4.16. This codebook was generated using a three level architecture containing three codebooks of size 8. MSVQ using a 4×3 block and a codebook size of 8192 is presented in Figure 4.17. This codebook was generated using a three level architecture containing three codebooks of size 16.

We also present one example of how the image develops through each stage of the MSVQ process. Figures 4.18-4.20 show each stage for the example presented in Figure 4.16. As we saw with TSVQ, the improvement is each stage is easy to see. The property of *graceful degradation* is also manifested by MSVQ, since the figures shown correspond to the lower bit rate images that would be produced if the channel codewords were truncated.

As with FSVQ and TSVQ, we can see that the performance of MSVQ depends strongly on the size of the codebook. The performance of MSVQ falls far short of the standard set by FSVQ as can be seen in the MSE comparison shown in Figure 4.21. The reason for this large degree of suboptimality can be seen in the structure of MSVQ. Consider a TSVQ structure in which we formed the data subsets for the next

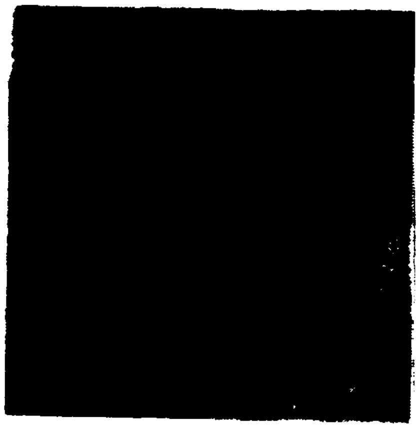

Figure 4.14 MSVQ Using a Size 16 Codebook and a 2x2 Block

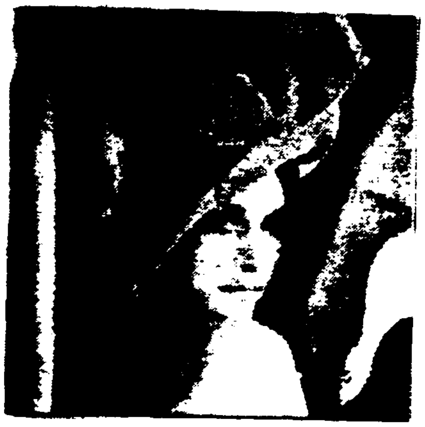

Figure 4.15 MSVQ Using a Size 64 Codebook and a 3x2 Block

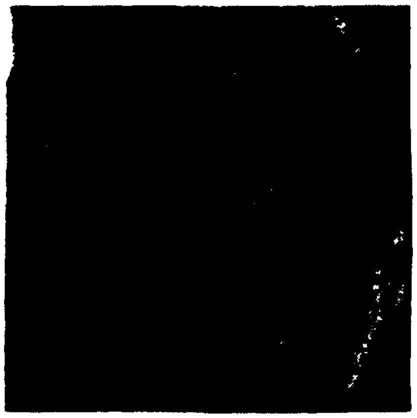

Figure 4.16 MSVQ Using a Size 512 Codebook and a 3x3 Block

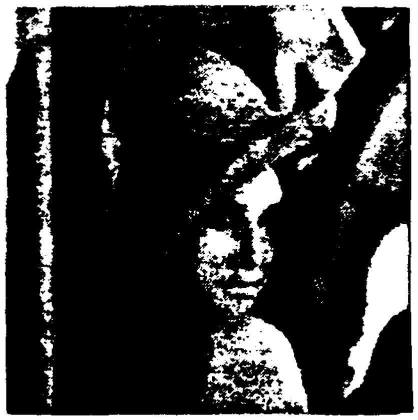

Figure 4.17 MSVQ Using a Size 4096 Codebook and a 4x3 Block

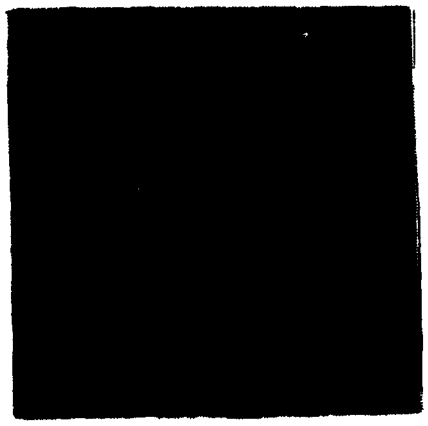

Figure 4.18 MSVQ Using 3x3 Block
First Stage

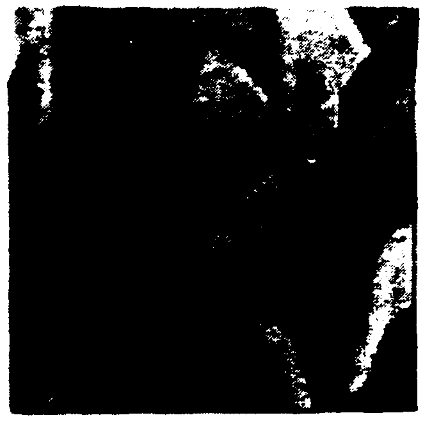

Figure 4.19 MSVQ Using 3x3 Block
Second Stage

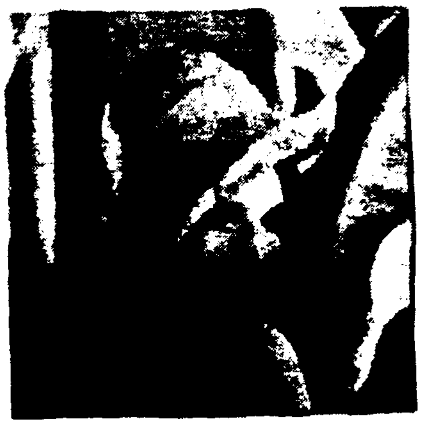

Figure 4.20 MSVQ Using 3x3 Block
Third Stage

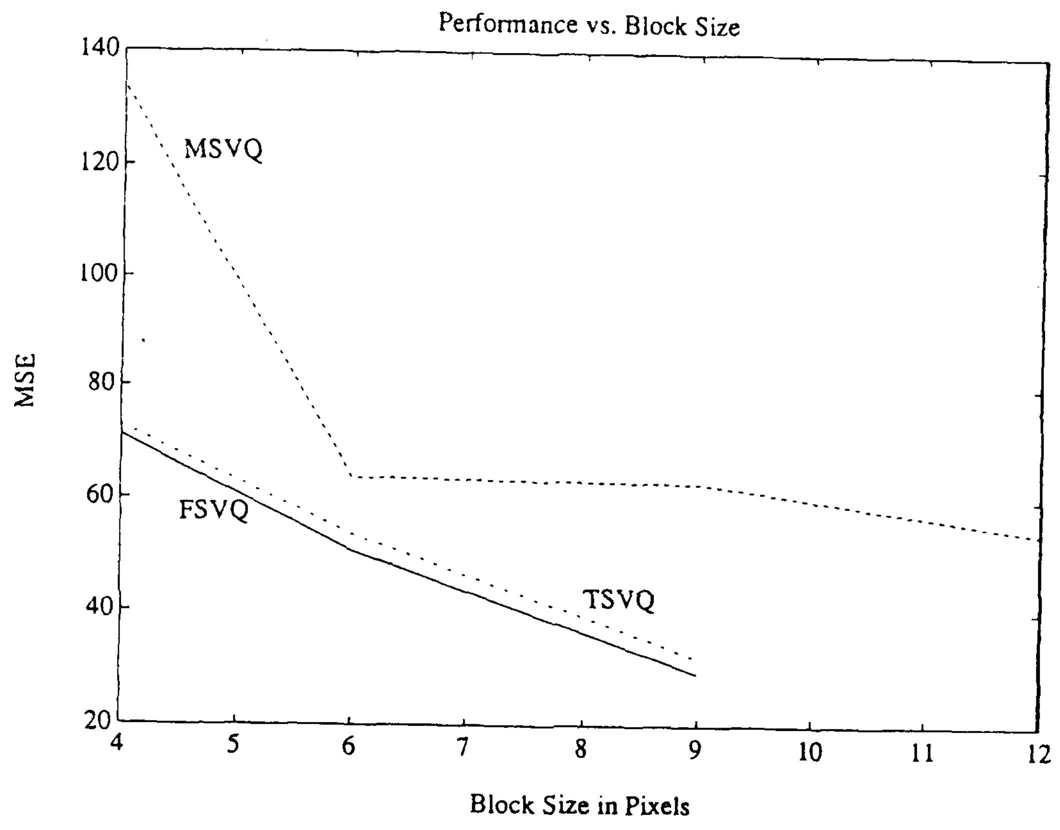

Figure 4.21 Performance vs. Block Size

level using the error vector instead of the original input vector. Since the vector quantization process is translation invariant, the performance of this new structure would be identical to the original TSVQ. We can also see that this structure is the same as MSVQ except that a different codebook is used to encode the error vectors for each branch of the tree. Thus MSVQ is equivalent to TSVQ if we assume that the probability distribution function which describes the distribution of the errors about each code vector on the same level of the tree is identical. That this assumption is far from the truth accounts for the relatively poor performance of the MSVQ algorithm.

Although the performance of MSVQ is poor relative to FSVQ and TSVQ for codebooks of the same size, MSVQ maintains several highly desirable features. We can see from Figure 4.22 that MSVQ provides a huge computational advantage for large codebooks. MSVQ also provides an extremely simple structure which would require only a small number of processing elements and would make hardware implementation much simpler. Finally, because MSVQ uses only one vector quantizer per level, the algorithm vastly reduces the amount of storage required for simulation and decreases the load on the transmission channel due to codebook transmission.

D. CLASSIFICATION VECTOR QUANTIZATION

The refinements to the basic FSCL algorithm that we have examined so far concentrate on reducing the computational cost of training the vector quantizer system. Our standard for performance in all cases has been the mean square error. Now we take a brief look at the subjective quality of the images produced. The most noticeable problem with each of the methods is the *staircase effect*. This is where an edge follows the outline of the blocks rather than the smooth edge of the original image as can be seen by examining the curve in the shoulder in Figures 4.3 and 4.5. This *staircase effect* follows the size of the block used in coding the image, and will thus

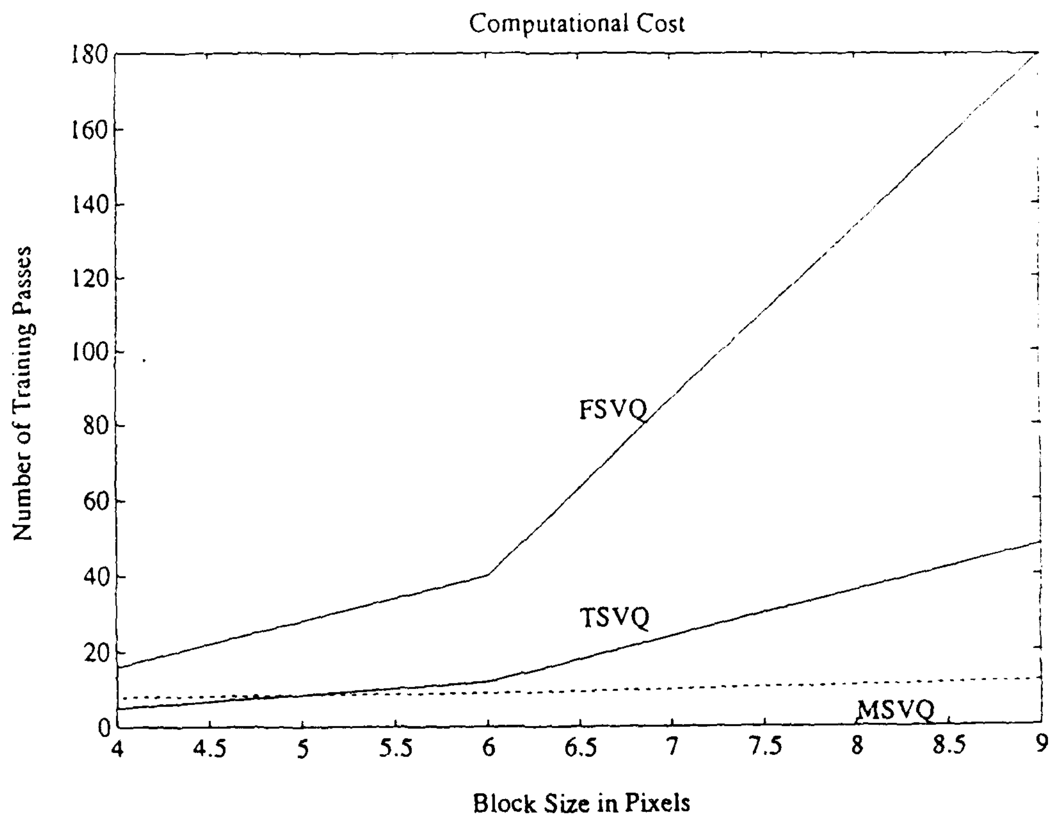

Figure 4.22 Computational Cost

become more and more noticeable as the block size is increased. This puts us in the uncomfortable situation of wanting to increase the block size to improve mean square error performance and at the same time wanting to reduce the block size to minimize this *staircase effect*. In order to solve this dilemma we need to examine the cause of this *staircase effect* and look at possible solutions.

One possible cause is that the codebook does not contain a sufficient variety of code vectors which represent blocks with edges. To examine this possibility, a codebook for a FSVQ with a 2×2 block size is presented in table 4.6. The four pixel values in each row constitute a code vector. We would expect a code vector which represents an edge to contain both high and low values, but upon examining the codebook in table 4.6, we see that the code vectors exhibit almost no structure and are certainly inadequate to represent all the possible edge configurations. To examine the reason for this under-representation of edge blocks, we introduce an edge detector which is used to indicate whether an edge appears somewhere in the block.

For each set of adjacent pixels in the block, we take the pixel values, m_1 and m_2 and form the ratio $\frac{|m_1 - m_2|}{\max(m_1, m_2)}$ and apply a threshold to determine if this is an edge block or a shade block. The results of applying this ratio to our test image is presented in Figure 4.23 for a block size of 2×2. The authors of [Ref. 16] chose a threshold of 0.4 to define an edge block. Applying this value gives us only 202 edge blocks out of a total of 16384 blocks in the image. Thus it appears that the problem with edges occurs because there are so few edge blocks in the image, and the poor representation of these blocks do not contribute significantly to the mean square error. So the root of the problem seems to be that the distortion measure, i.e., mean square error, fails to take into account the perceptual importance of the edge blocks. This leaves two basic solutions: change to a more complicated, perceptually based distortion measure, or divide the problem by using separate vector quantizers

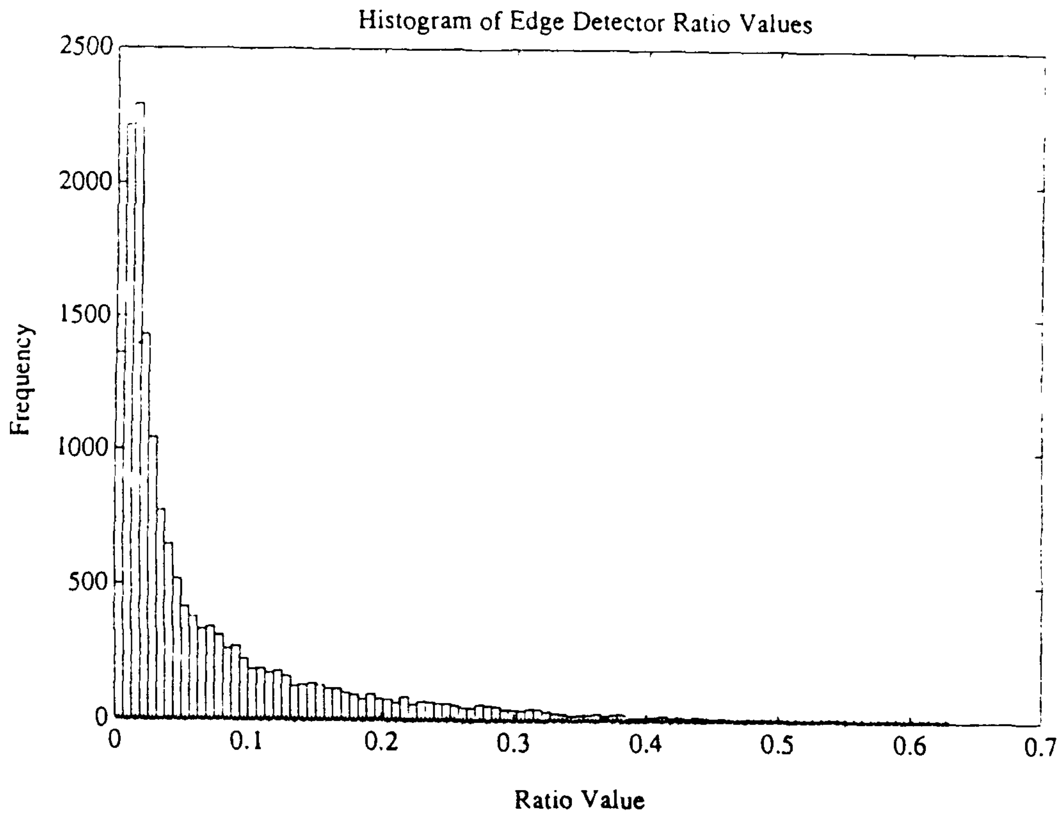

Figure 4.23 Histogram of Edge Detector Ratio Values

on the edge and shade blocks.

The technique of Classification Vector Quantization (CVQ) [Ref. 16] (See Figure 4.13) uses the second method discussed above to improve the subjective quality of the reconstructed image. The image is divided into blocks as before, but now we apply the edge detector and use a threshold to separate the image into two data sets one containing the edge blocks and the other containing the shade blocks. These two data sets are then applied separately to a FSCL vector quantizer which is trained until convergence. The two resulting codebooks are then concatenated to form an overall codebook which emphasizes the edge blocks . The amount of emphasis given to the edge blocks depends on the sizes of the codebooks allocated to the edges and shades. For example, a codebook size of 64 could be divided into 48 shade code vectors and 16 edge code vectors. This would give the edges more emphasis than the original technique. Even further emphasis would be obtained if we used 32 shade and 32 edge code vectors instead.

The simulation results for CVQ are presented in Figures 4.24-4.26. As before a single test image of 256×256 pixels was used, and all test cases were conducted at 1 bit/pixel. Figure 4.24 shows CVQ using a 2×2 block and a size 16 codebook consisting of 8 edge and 8 shade code vectors. Figure 4.25 shows CVQ using a 3×2 block and a size 64 codebook consisting of 32 edge and 32 shade pixels. Figure 4.26 shows CVQ using a 3×3 block and a size 512 codebook consisting of 384 edge and 128 shade code vectors. For the size 16 case (Figure 4.24) we can see that the codebook is just too small to represent shades and edges well. The lack of enough shade code vectors to cover the common grey levels is evident, and the few edge code vectors are not enough to show much improvement over FSVQ. In the size 64 case (Figure 4.25) we start to see some substantial improvement in the reproduction of the edges with very little degradation in other areas of the image. Finally, for the size 512 case

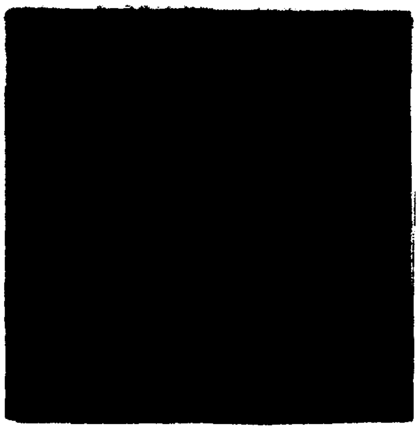

Figure 4.24 CVQ Using a Size 16 Codebook and a 2x2 Block

Figure 4.25 CVQ Using a Size 64 Codebook and a 3x2 Block

Figure 4.26 CVQ Using a Size 512 Codebook . ıd a 3x3 Block

(Figure 4.26). CVQ is substantially better in a subjective sense, and for the larger codebooks and larger block sizes it is slightly better than FSVQ in the mean square error sense (See Figure 4.27). It is surprising that any method could surpass the performance of FSVQ since we belieived this method to be optimal in a mean square sense, but this effect probably stems from the fact that FSVQ converges very slowly, and the test cases were not run a sufficient number of training passes to reach the final value.

A secondary benefit of applying the CVQ technique is an enormous computational savings over FSVQ. This occurs because the code vectors for edge and shade pixels appear to converge at different rates. The shade code vectors have a very simple structure and therefore converge quickly, while the edge code vectors have a complex structure and converge slowly. In FSVQ, we use a single codebook and thus all code vectors are run through the data set the same number of times. So long after the shade code vectors have converged, we continue to waste computational time updating them. In CVQ, we avoid this problem, and we are then able to concentrate our computational efforts on the difficult part of the problem. Also as we have seen with TSVQ, a data set which has a large amount of structure makes the FSCL algorithm converge more quickly. The CVQ method accomplishes this by splitting the original data set into shade and edge blocks which further improves convergence speed. As we can see in Figure 4.28, CVQ has a huge computational advantage over FSVQ as well as better performance for large codebooks.

TABLE 4.3: Number of Codebooks Required

			Codebooks	
R	Block Size	Codebook Size	TSVQ	MSVQ
(2,2)	2×2	16	5	2
(3,2)	3×2	64	9	2
(3,3,3)	3×3	512	73	3

TABLE 4.4: Code Vector Storage Requirements

			Code Vectors		
R	Block Size	Codebook Size	FSVQ	TSVQ	MSVQ
(2,2)	2×2	16	16	20	8
(3,3)	3×2	64	64	72	16
(3,3,3)	3×3	512	512	584	24

TABLE 4.5: Channel Load of Codebook Transmission (bits/pixel)

			Channel Load (bits/pixel)		
R	Block Size	Codebook Size	FSVQ	TSVQ	MSVQ
(2,2)	2×2	16	0.008	0.010	0.004
(3,3)	3×2	64	0.047	0.053	0.012
(3,3,3)	3×3	512	0.563	0.642	0.026

TABLE 4.6: Example Codebook

Codeword	Pixel 1	Pixel 2	Pixel 3	Pixel 4
1	159	167	186	193
2	200	200	202	202
3	104	105	123	127
4	86	86	87	87
5	223	223	224	223
6	133	126	107	104
7	217	217	217	217
8	137	137	141	141
9	208	209	209	209
10	231	231	231	231
11	175	174	175	175
12	160	159	156	156
13	99	99	99	99
14	240	240	240	240
15	189	189	190	190
16	204	199	177	166

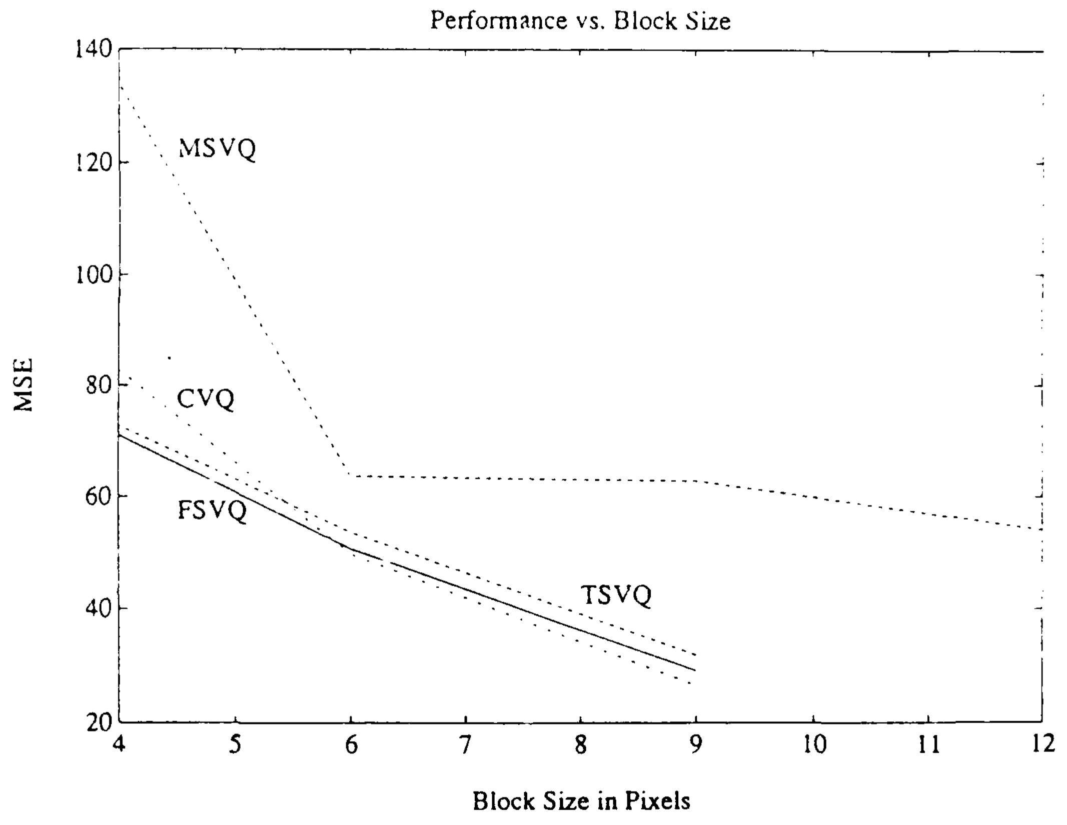

Figure 4.27 Performance vs. Block Size

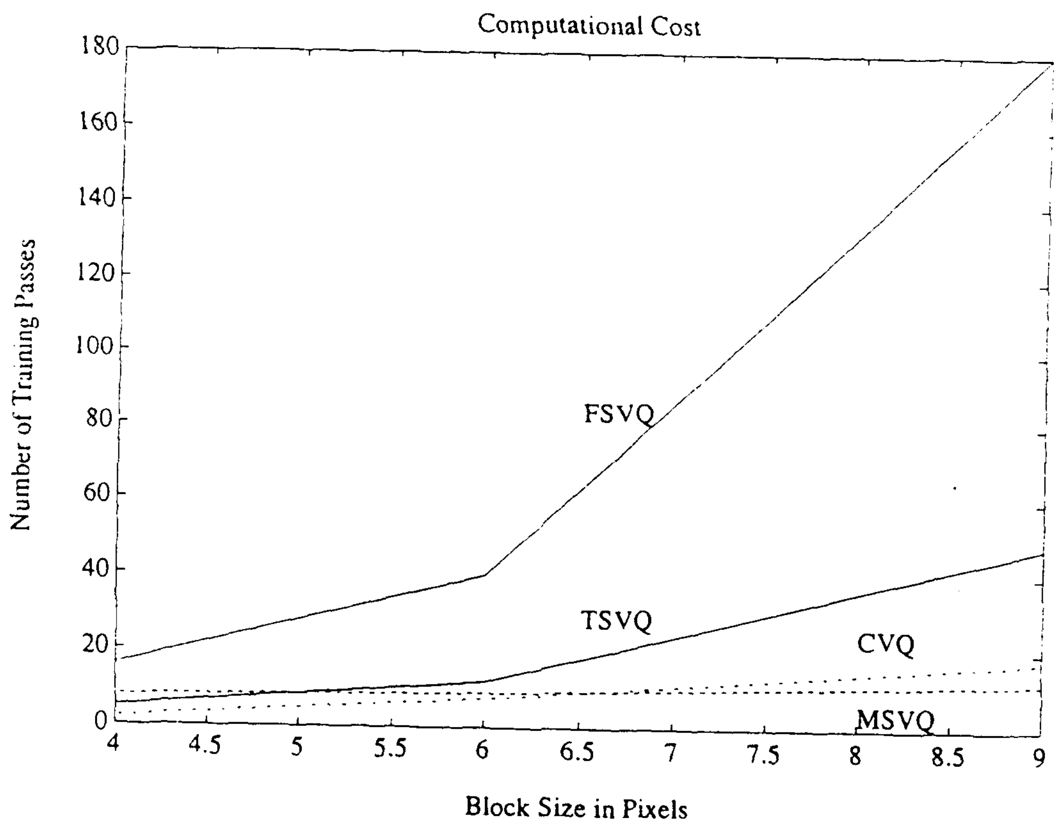

Figure 4.28 Computational Cost

V. CONCLUSIONS

In this thesis we examined some existing algorithms to implement vector quantization using neural networks. We also applied three techniques to improve performance and reduce computational cost. In the previous chapter we presented each technique separately. Here we will compare the relative performance of each of the three algorithms. Since each algorithm has its strengths and weaknesses, we also make suggestions about the likely situations where each of these techniques may be appropriate.

First let us discuss image reproduction quality. It can seen from Figure 4.27 that for a given block size, FSVQ, TSVQ, and CVQ all offer a similar level of performance in a mean square sense, while MSVQ performs noticeably worse. To compare performance in a subjective sense, we present the best results obtained for each technique in the following figures. Figure 5.1 shows the FSVQ algorithm using a 3×3 Block. Figure 5.2 shows TSVQ using a 3×3 block. Figure 5.3 shows MSVQ using a 4×3 Block, and Figure 5.4 shows CVQ using a 3×3 Block. Here we see that CVQ has a small advantage over FSVQ and TSVQ in a subjective sense, and MSVQ is again noticeably worse.

Now we examine the issue of computational cost. We can see from Figure 4.28 that for a given block size, FSVQ has the highest computational cost, TSVQ is the next highest, and CVQ and MSVQ have very similar and much smaller computational costs. Perhaps a better way to rate the computational cost is to relate it to performance. Figure 5.5 shows the relationship between cost and performance for each test case performed. Algorithms that are most desirable are represented by points in the lower left portion of the graph. We can see that the best combination of reproduction

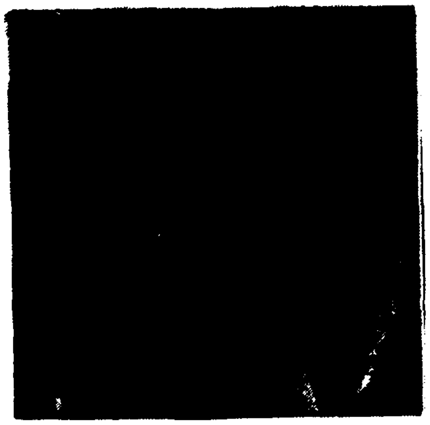
Figure 5.1 FSVQ

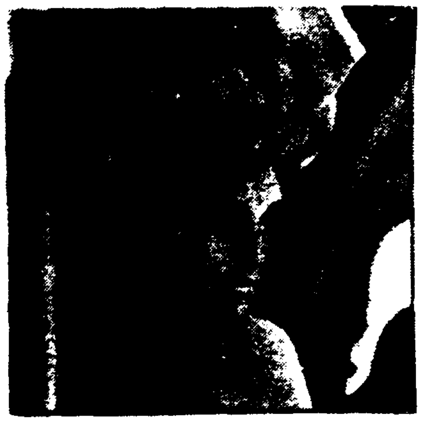
Figure 5.2 TSVQ

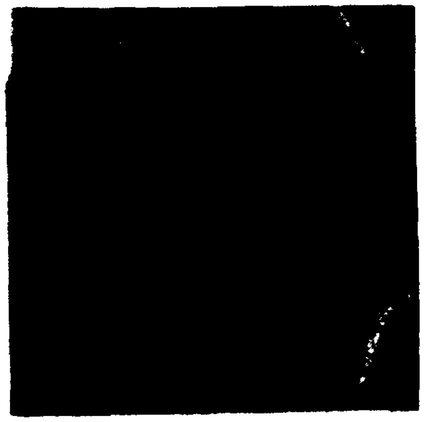
Figure 5.3 MSVQ

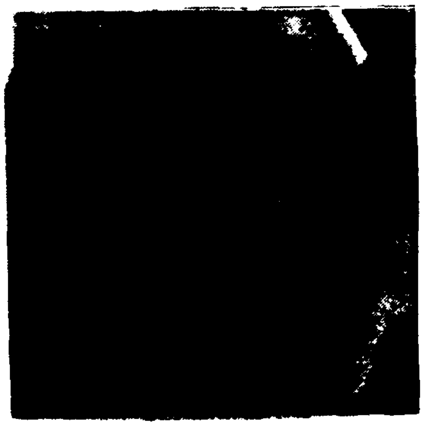
Figure 5.4 CVQ

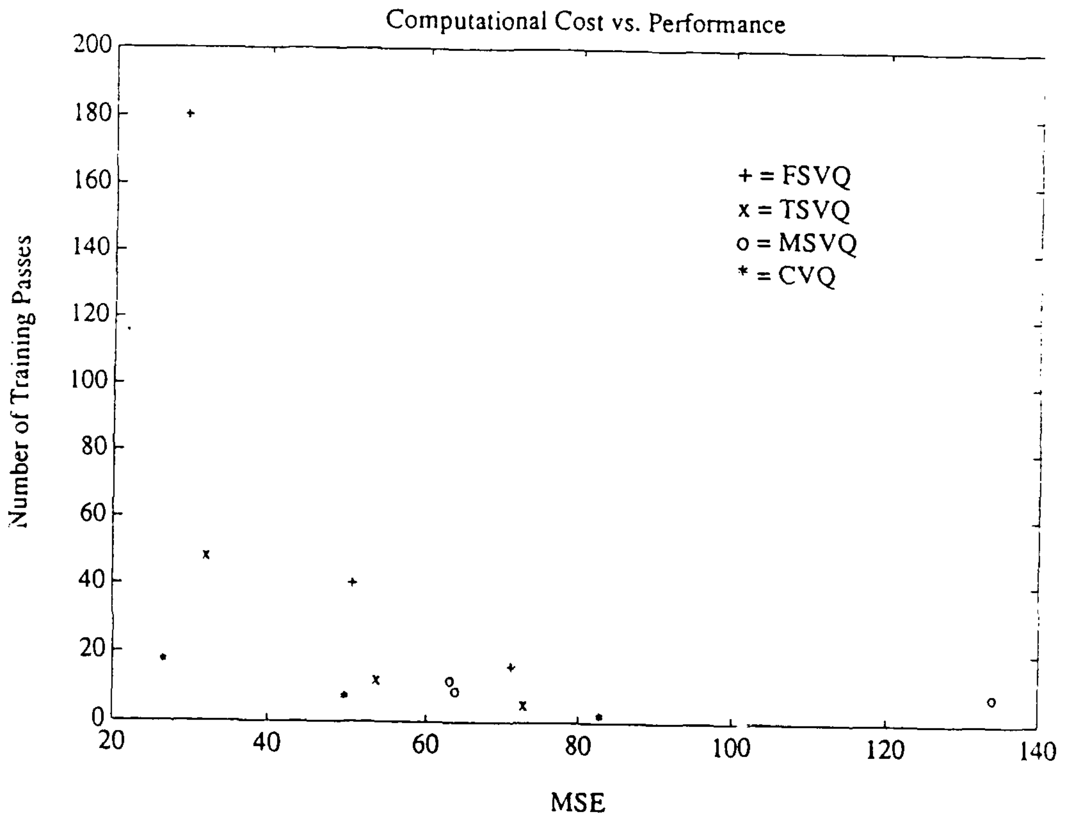

Figure 5.5 Computational Cost vs. Performance

quality and computational cost is given by the CVQ algorithm.

Although CVQ offers the best reproduction quality even when computational cost is considered, the other two algorithms presented have advantages of their own. Both MSVQ and TSVQ offer a huge savings in the number of processing elements required because of the linear structure each displays. Thus for a hardware implementation these two techniques should be considered. Also we have seen that for large code book sizes the load on the channel due to code book transmission becomes significant. So if our application requires an extremely large code book the MSVQ algorithm must be considered as it is able to form a large code book with very little load on the channel (See Table 4.5).

This research has shown that neural networks can be very effective in the implementation of vector quantization. With the application of algorithms such as CVQ, TSVQ, and MSVQ, we can improve the performance of neural network vector quantizers and make application of the vector quantization technique more practical.

A. ADDITIONAL WORK

Research is planned in the area of adaptive filters in an effort to improve the convergence speed of the FSCL algorithm. In addition, it is planned to investigate other current vector quantization techniques and determine if neural network vector quantizers can be improved by their application. After these steps are completed, an effort to combine several of the techniques chosen will be conducted in the hope of further improving overall performance. Finally, we plan to apply the techniques in this thesis to the coding of speech data.

APPENDIX A: PROGRAM DETAILS

This appendix contains the program flowcharts and listings for each of the algorithms in the thesis. Figures A.1-A.4 show the flowcharts, and the program listings follow.

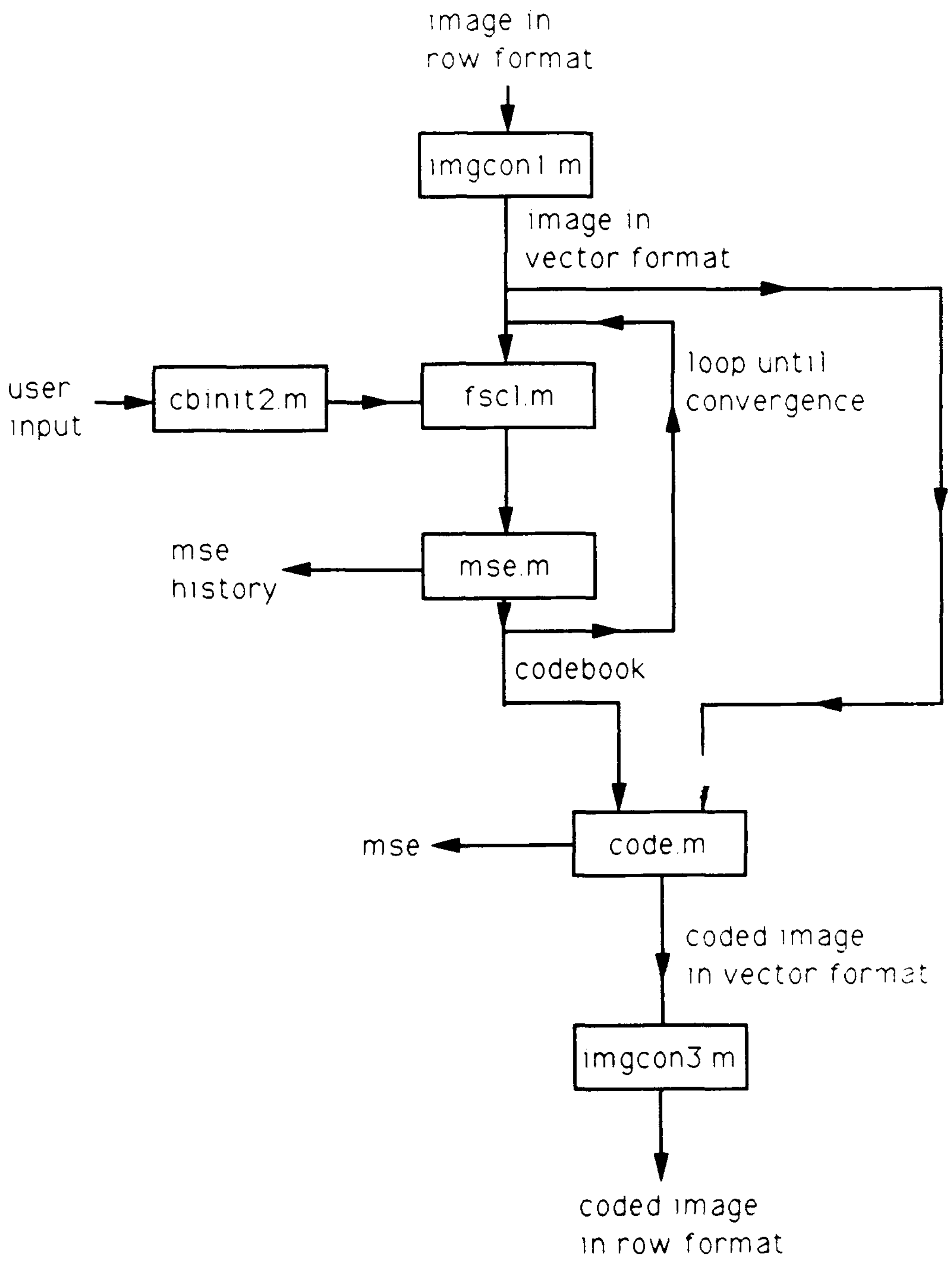

Figure A.1 Basic FSCL Algorithm

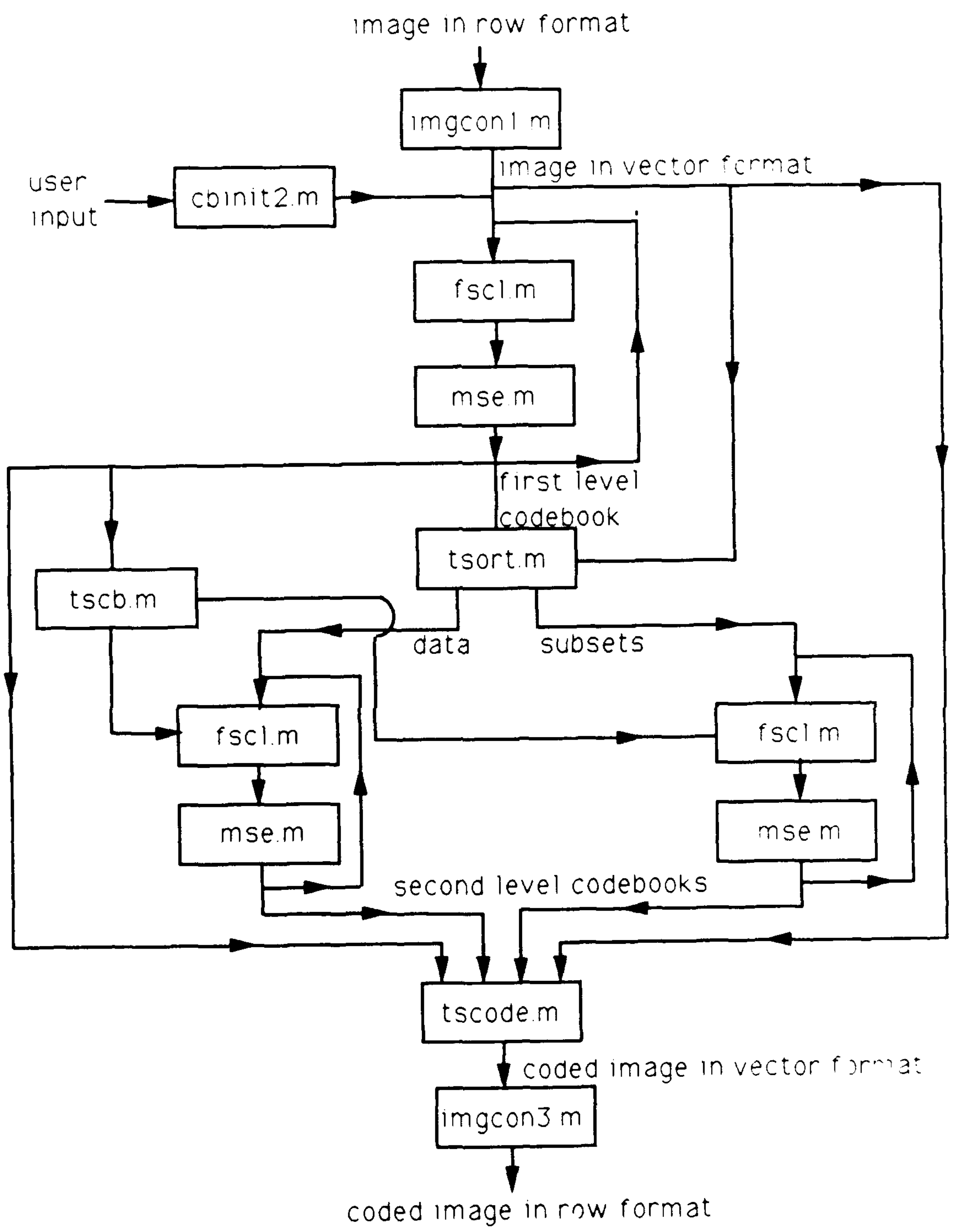

Figure A.2 Tree Search Algorithm

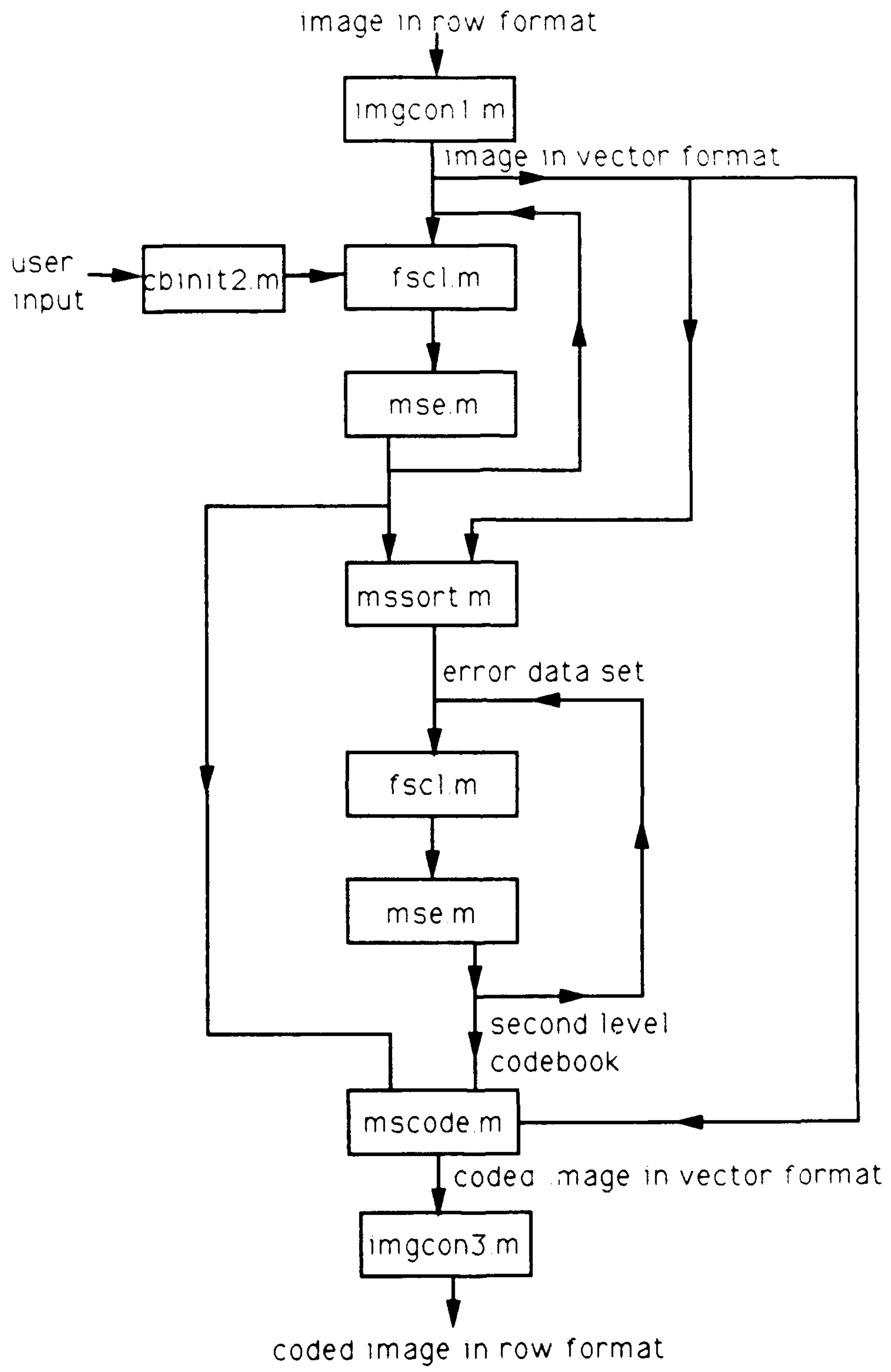

Figure A.3 Multi Stage Algorithm

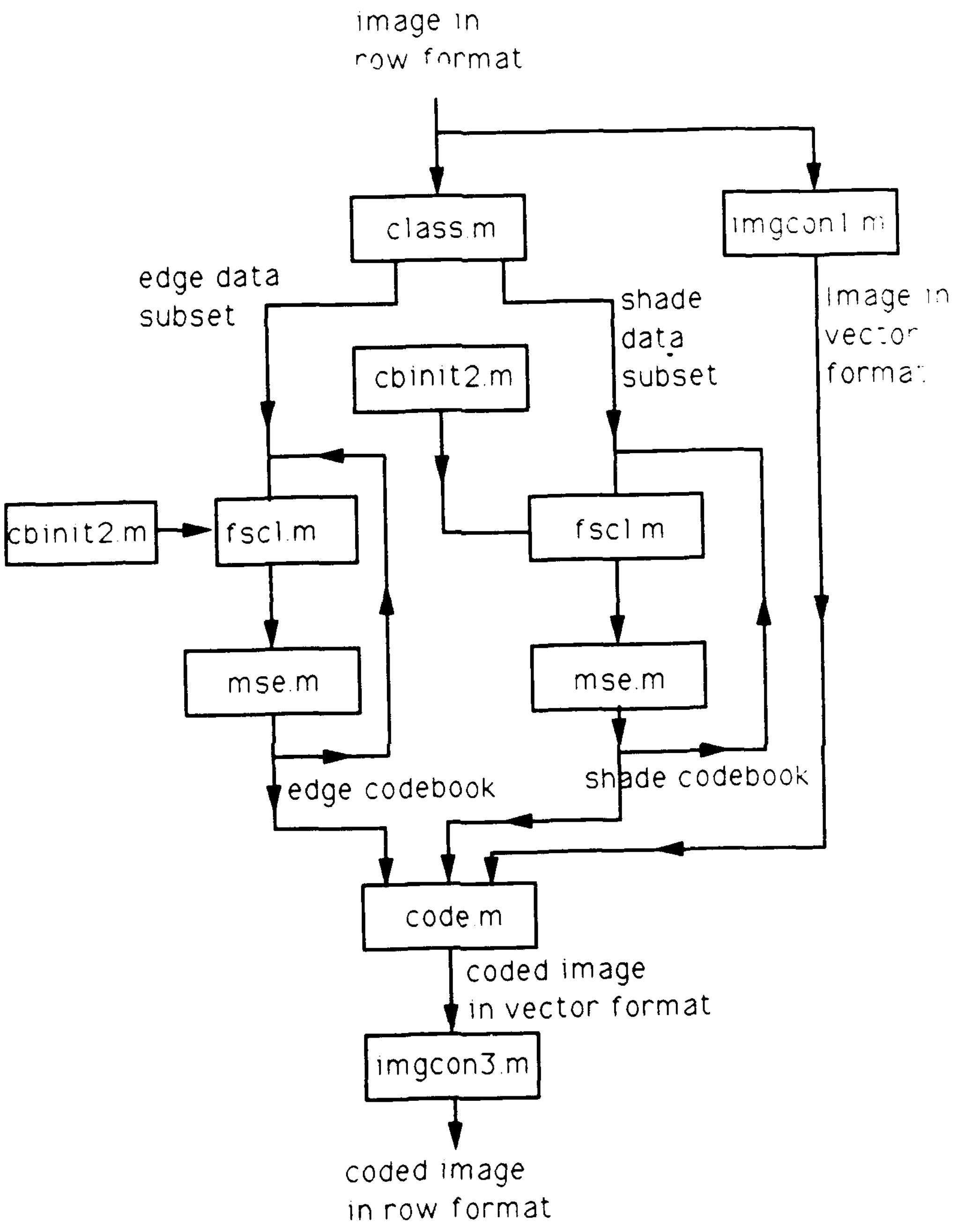

Figure A.4 Classification Algorithm

```
function x=imgconl(y)

% program to convert an array of image data into vector format using
% blocks of arbitrary size

% input variable y = subject image in row format

% output variable x = subject image in vector format

% local variables N = vector which stores the dimensions of y
                  n2 = height of desired block input by user
                  nl - width of block input by user
                  nla = number of blocks to process in horizontal direction
                  n2a = number of blocks to process in vertical direction
                  k = index to track number of blocks processed
                  il = vertical placekeeper in subject image
                  jl = horizontal placekeeper in subject image
                  z = temporary storage for desired block

N=size(y);                               % initialize dimensions of input image
n2=input('height of block   ');          % get height og block from user
nl=input('width of block    ');          % get width of block from user
nla=floor(N(1)/nl);                      % find # of blocks to process horiz.
n2a=floor(N(2)/n2);                      % find # of blocks to process vertic.
x=zeros(nl*n2,nla*n2a);                  % initialize output
for i=1:n2a                              % main loop: move vertically in image
   il=(i-1)*n2+1                         % set vertical placekeeper
   for j=1:nla                           % inner loop: move horizontally in image
     k=(i-1)*nla+j;                      % track number of blocks processed
     jl=(j-1)*nl+1;                      % set horizontal placekeeper
     z=y(il:il+n2-1,jl:jl+nl-1);         % get desired block from image
     x(:,k)=z(:);                        % make conversion from block to vector
   end
end
```

```
function [w,u]=cbinit2(x)

% program to initialize the code book

% input variable x = data set in vector format

% output variable w = initial code book
%                 u = initialized friquency vector

% local variables N = desired number of code words
%                 Nx = number of data vectors

% this program initializes the codebook by randomly selecting data vectors from the
% subject data set. It also sets up hte initial frequency vector for the codebook
% with all values initialized to 1.


N=input('number of code words   ');

rand('uniform')
Nx=max(size(x));
for i=1:N
   w(:,i)=x(:,ceil(Nx*rand(1)));
end
u=ones(1,N);
```

```
function [w,u]=fscl(x,w,u)

% program to implement frequency sensitive competitive learning
%
% input variables x =  subject data set arranged into vectors of appropriate size
%                 w =  existing weight matrix
%                 u =  existing win frequency vector
%
% output variables w - updated weight matrix
%                  u = updated win frequency vector
%
% local variables nx = size of data vectors
%                 Nx = number of data vectors   Caution: Nx must be > nx
%                 N  = vector containing the size and number of weight vectors in w
%                 y  = ones vector used to set up comparison of distances
%                 d  = vector which stores the distance for each code vector
%                 md = the minimum distance contained in d
%                 iw = the index of code vector with minimum distance
%                 ep = learning rate
%
% This program conducts a single pass through data set x using the FSCL algorithm. The
% weight matrix, w, and win frequency vector, u, are updated and passed back to the calling
% routine.

nx=min(size(x));                              % initialize size of data vector
Nx=max(size(x));                              % initialize number of data vectors
N=size(w);                                    % initialize dimensions of weight matrix
y=ones(1,N(2));                               % initialize ones vector

for k=1:Nx                                    % main loop: perform once for each data vector
    d=sum((x(:,k)*y-w).^2);                   % calculate distance for each code vector
    d=d.*u;                                   % apply fairness function to distance
    [md,iw]=min(d);                           % find minimum distance
    ep=0.01*exp(-u(iw)/10000);                % determine learning rate for nearest
                                              % code vector
    w(:,iw)=w(:,iw)+ep*(x(:,k)-w(:,iw));      % update weight vector for nearest code vector
    u(iw)=u(iw)+1;                            % update number of wins for nearest code vector
end
```

```
function m=mse(x,w,m)

% program to measure mean square error of codebook

% input variables x =  subject data set arranged into vectors of appropriate size
%                 w =  weight matrix of codebook to be measured
%                 m =  record of mse for previous versions of code book
%
% output variable m = updated record of mse measurements
%
% local variables Nx = number of data vectors   Caution: Nx must be > nx
%                 N  = vector containing the size and number of weight vectors in w
%                 y  = ones vector used to set up comparison of distances
%                 d  = vector which stores the distance for each code vector
%                 msel = accumulator for current mse

% this program makes a single pass through the data set in order to measure the mse.
% the mse is then appended to an existing vector, m, which has the mse record for
% each iteration of the codebook

N=size(w);                       % initalize size of weight matrix
Nx=max(size(x));                 % initialize number of data vectors
msel=0;                          % initialize mse accumulator
y=ones(1,N(2));                  % initialize ones matrix

for k=1:Nx                       % main loop : execute once for each data vector
   d=sum((x(:,k)*y-w).^2);       % calculate for each weight vector
   msel=msel+min(d);             % increment mse accumulator
end
msel=msel/(Nx*N(1));             % normalize mse
m=[m,msel];                      % append new mse value to previous record
```

```
function [z,mse]=code(x,w)
```

```
Sep 20 11:03 1991  cbinit2.m Page 1

function [w,u]=cbinit2(x)

% program to initialize the code book

% input variable x = data set in vector format

% output variable w = initial code book
%                 u = initialized frrquency vector

% local variables N = desired number of code words
%                 Nx = number of data vectors

% this program initializes the codebook by randomly selecting data vectors from the
% subject data set. It also sets up hte initial frequency vector for the codebook
% with all values initialized to 1.


N=input('number of code words   ');

rand('uniform')
Nx=max(size(x));
for i=1:N
   w(:,i)=x(:,ceil(Nx*rand(1)));
end
u=ones(1,N);
```

```
function x=imgcon3(y)

% program to convert an image in vector format to an image in row
% format with an arbitrary vector size

% input variable y = subject image in vector format

% output variable x = subject image in row format

% local variables N = vector which stores the dimensions of y
                  n2 = height of block input by user
                  n1 = width of block input by user
                  n3 = size of desired output image
                  n1a = number of blocks to process in horizontal direction
                  n2a = number of blocks to process in vertical direction
                  k = index to track number of blocks processed
                  i1 = vertical placekeeper in subject image
                  j1 = horizontal placekeeper in subject image
                  z = temporary storage for desired block


N=size(y);                          % initialize dimensions of input image
n2=input('height of block    ');    % get height of block from user
n1=input('width of block     ');    % get width of block from user
n3=input('size of output image      ');  % get desired output image size
n1a=floor(n3/n1);                   % find # of blocks in vert. direction
n2a=floor(n3/n2);                   % find # of blocks in horiz. direction
x=zeros(n3,n3);                     % initialize output image
for i=1:n2a                         % main loop : move vertically
   i1=(i-1)*n2+1                    % set vertical placekeeper
   for j=1:n1a                      % inner loop : move horizontally
     k=(i-1)*n1a+j;                 % update number of blocks processed
     j1=(j-1)*n1+1;                 % set horizontal placekeeper
     z=zeros(n2,n1);                % initaiize temporary storage
     for l=1:n1                     % loop to convert vector to block
       m=(l-1)*n2+1;                % find section of vector to process
       z(:,l)=y(m:m+n2-1,k);        % get segment of vector
     end
     x(i1:i1+n2-1,j1:j1+n1-1)=z;    % put completed block into image
   end
end
```

```
% program to initialize the codebook for TSVQ

% variables x1,x2,... = input data sets in vector form
%           w1,w2,... = initial code books
%           u1,u2,... = initialized frequency vectors
%           N = desired number of code words
%           Nx = number of data vectors in data set being processed

n=input('size of input vector   ');            % get size of input vector
N=input('number of codewords   ');             % get desired number of codewords
nb=input('number of branches in tree  ');      % get number of branches in tree

% this program constructs the code book initialization for TSVQ by randomly
% chosing input data vectors from each data set

rand('uniform')                  % set up random number generator

for p=1:nb          % main loop execute once for each branch of tree
    eval(['Nx=size(x',int2str(p),');']);  % get size of current data set
    Nx=Nx(2);
    for q=1:N        % inner loop : choose N random vectors from data set
       m=ceil(Nx*rand(1,1));     % select random number
       eval(['w',int2str(p),'(:,q)=x',int2str(p),'(:,m);'])
       % place selected vector in appropriate code book
    end
    eval(['u',int2str(p),'=ones(1,N);']);  % initialize frequency counter
    eval(['m',int2str(p),'=[ ];']);         % initialize mse history
end
```

```
% program to sort vector for tree searched code

% variables Nx = number of data vectors   Caution: Nx must be > nx
%           N  = vector containing the size and number of weight vectors in w
%           y  = ones vector used to set up comparison of distances
%           d  = vector which stores the distance for each code vector
%           md = minimum distance from data vector to a code word
%           iw = index of minimum distance in d
%           x =  subject data set arranged into vectors of appropriate size
%           w =  weight matrix of codebook to be used for sorting
%           x1,x2,... = data sets of vectors for use in next stage
%           count = vector to track size of output data sets

% this program performs the sorting of hte input data set for use by the sceond % level of t

N=size(w);                      % initialize dimensions of w
Nx=max(size(x));                % initialize number of input vectors
mse=0;                          % initialize mse
y=ones(1,N(2));                 % initialize ones vector

for k=1:N(2)        % this loops initializes the output data sets
    eval(['x',int2str(k),'=zeros(N(1),Nx/4);']);
    count(k)=0;     % initialize size of output data sets
end

for k=1:Nx                      % main loop : execute once for each input vector
   d=sum((x(:,k)*y-w).^2);      % calculate distances for each code vector
   [md,iw]=min(d);              % find closest code vector
   count(iw)=count(iw)+1;       % update size of output data set chosen
   eval(['x',int2str(iw),'(:,count(iw))=x(:,k);']);  % update output data set
   if rem(k,1000)==0, k, end    % update progress to screen
end

for k=1:N(2)     % this loop truncates the output data sets to eliminate
                 % the unused portion of the allocated space
    eval(['x',int2str(k),'=x',int2str(k),'(:,1:count(k));']);
end
```

```
% program to code an image for tsvq

% variables x = data set with image in vector format
%           w = weight matrix for vector quantizer at first level
%           w1,w2,... = weight matrices for vector quantizer at second level
%           wa = weight matrix chosen for use at second level
%           z  = approximate image produced by coding in vector format
%           mse = mean square error of approximation, z
%           N  = vector containing size and number of weight vectors in w
%           Nx = number of data vectors
%           y  = ones vector used to set up comparison of distances
%           d  = vector which stores the distance for each code vector
%           d2 = vector distances for code book at second level
%           md = the minimum distance contained in d
%           md2 =  the minimum distance contained in d2
%           iw = the index of the code vector with minimum distance
%           iw2 = index of closest code vector in level two

% this progrm performs coding for a two level TSVQ. The input and output
% images are both in vector format


N=size(w1);             % initialize dimensions of w
Nx=max(size(x));        % initialize number of input data vectors
mse=0;                  % initialize mse
y=ones(1,N(2));         % initialize ones vector
z=zeros(N(1),Nx);       % initialize output image

for k=1:Nx                    % main loop : execute once for each input vector
   d=sum((x(:,k)*y-w).^2);    % find distances for code book at first level
   [md,iw]=min(d);            % find closest code vector at first level
   eval(['wa=w',int2str(iw),';']);  % pick weight matrix to be used at level two
   d2=sum((x(:,k)*y-wa).^2);   % find distances for code book at level two
   [md2,iw2]=min(d2);          % find closest code vector at level two
   z(:,k)=wa(:,iw2);           % place approximation in output image
   mse=mse+sum((x(:,k)-z(:,k)).^2);   % increment mse
   if rem(k,1000)==0, k, end          % update progess to screen
end
mse=mse/(Nx*N(1));                    % normalize mse
```

Sep 20 11:07 1991 mssort.m Page 1

```
function z=mssort(x,w)

% program to set up multi stage vq

% input variables x =  subject data set arranged into vectors of appropriate size
%                 w =  weight matrix of codebook to be used for sorting
%
% output variable z = data set of error vectors for use in next stage
%
% local variables Nx = number of data vectors   Caution: Nx must be > nx
%                 N  = vector containing the size and number of weight vectors in w
%                 y  = ones vector used to set up comparison of distances
%                 d  = vector which stores the distance for each code vector
%                 md = minimum distance from data vector to a code word
%                 iw = index of minimum distance in d

% this program takes a data set and a code book and performs on pass through each
% data vector, finding the closest code vector and calculating and storing the
% error. This new data set is used for the next stage in Multi Stage Vector
% Quantization.

N=size(w);                          % initalize number and size of weight vectors
Nx=max(size(x));                    % initialize number ofdata vectors
y=ones(1,N(2));                     % initialize ones vector
z=zeros(N(1),Nx);                   % initialize error data set

for k=1:Nx                          % main loop : execute once for each data vector
   d=sum((x(:,k)*y-w).^2);          % calulate distance for each code word
   [md,iw]=min(d);                  % find the minimum distance
   z(:,k)=x(:,k)-w(:,iw);           % calculate and store the error vector
   if rem(k,1000)==0, k, end        % update progress every 1000 data vectors
end
```

```
function [z,mse]=mscode(x,w,w1)

% program to code an image for ms vq

% input variables x = data set with image in vector format
%                 w = weight matrix for vector quantizer at first level
%                 w1 = weight matrix for vector quantizer at second level

% output variables z  = approximate image produced by coding in vector format
%                   mse = mean square error of approximation, z

% local variables N  = vector containing size and number of weight vectors in w
%                 Nx = number of data vectors
%                 x2 = data set containing first level error
%                 y  = ones vector used to set up comparison of distances
%                 d  = vector which stores the distance for each code vector
%                 d2 = vector distances for code book at second level
%                 md = the minimum distance contained in d
%                 md2 =  the minimum distance contained in d2
%                 iw = the index of the code vector with minimum distance
%                 iw2 = index of closest code vector in level two

% this program performs coding for the MSVQ algorithm. This version is
% to a two level architecture. The input and output image are both in
% vector format.

N=size(w);                          % initialize dimensions of w
Nx=max(size(x));                    % initialize number of data vectors
mse=0;                              % initialize mse
y=ones(1,N(2));                     % initialize ones vector
z=zeros(N(1),Nx);                   % initialize output image

for k=1:Nx                          % main loop : execute once for each data vector
   d=sum((x(:,k)*y-w).^2);          % find distances for first level code book
   [md,iw]=min(d);                  % find closest code vector on first level
   x2=x(:,k)-w(:,iw);               % form first level error
   d2=sum((x2*y-w1).^2);            % find distances for second level code book
   [md2,iw2]=min(d2);               % find closest code vector on second level
   z(:,k)=w(:,iw)+w1(:,iw2);        % form second approximation to input vector
   mse=mse+sum((x(:,k)-z(:,k)).^2);   % increment mse
   if rem(k,1000)==0, k, end        % update progress to screen
end
mse=mse/(Nx*N(1));                  % normalize mse
```

```
function [x1,x2]=class1(y)

% program to convert an array for use in classified vq

% input variable y = subject image in row format
% output variables x1 = vector format array of edge blocks
                   x2 = vector format array of shade blocks
% local variables N = vector which stores the dimensions of y
%                 n2 = height of desired block input by user
%                 n1 = width of block input by user
%                 n1a = number of blocks to process in horizontal direction
%                 n2a = number of blocks to process in vertical direction
%                 k = index to track number of blocks processed
%                 i1 = vertical placekeeper in subject image
%                 j1 = horizontal placekeeper in subject image
%                 z = array used to evaluate edge detector ratio

% this program takes an image in row format applies an edge detector, and
% outputs two data sets in vector format. The first data set consists of
% the edge blocks, and the second consists of the shade pixels.


N=size(y);
n2=input('height of block   ');
n1=input('width of block    ');
n1a=floor(N(1)/n1);
n2a=floor(N(2)/n2);
x1=zeros(n1*n2,n1a*n2a);
x2=x1;
count1=0;
count2=0;
for i=1:n2a
   i1=(i-1)*n2+1
   for j=1:n1a
     k=(i-1)*n1a+j;
     j1=(j-1)*n1+1;
     z=y(i1:i1+n2-1,j1:j1+n1-1);
     z1=z(:);
     z2(1)=(z1(1)-z1(2))/max(z1(1),z1(2));
     z2(2)=(z1(1)-z1(3))/max(z1(1),z1(3));
     z2(3)=(z1(1)-z1(4))/max(z1(1),z1(4));
     z2(4)=(z1(2)-z1(3))/max(z1(2),z1(3));
     z2(5)=(z1(2)-z1(4))/max(z1(2),z1(4));
     z2(6)=(z1(3)-z1(4))/max(z1(3),z1(4));
     if max(abs(z2)) > 0.4
        count1=count1+1;
        x1(:,count1)=z1;
     else
        count2=count2+1;
        x2(:,count2)=z1;
     end
   end
end
x1=x1(:,1:count1);
x2=x2(:,1:count2);
```

REFERENCES

1. A. K. Jain. "Image Data Compression: A Review." *Proceedings of the IEEE*. , 349-389. 1981.

2. R. M. Gray. "Vector Quantization." *IEEE ASSP Magazine*.Vol. **1**. 4-29. Jan. 1984.

3. C. E. Shannon. "Coding theorems for a discrete source with a fidelity criterion." *IRE National Convention Record. Part 4*. 142-163. 1959.

4. S. P. Lloyd. "Least squares quantization in PCM." Bell Laboratories Technical Note. 1957.

5. H. Abut (Ed.). "Vector Quantization". IEEE Press. 72-86. 1990.

6. Y. Linde. A. Buzo. and R. M. Gray. "An algorithm for vector quantizer design." *IEEE Transactions on Communications*. COM-28. 84-95. January 1980.

7. D. O'Shaughnessy. "Speech Communication". Addison-Wesley. 313-323. 1987.

8. R. P. Lippman. " An Introduction to Computing with Neural Nets." *IEEE ASSP Magazine*. 4-22. April 1987.

9. D. E. Rumelhart. G. E. Hinton. and R. J. Williams. " Learning Internal Representations by Error Propagation." *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol 1: Foundations*. MIT Press. 1986.

10. T. Kohonen. "Self-Organization and Associative Memory." Spring Verlag. 1984.

11. D. DeSieno. " Adding a conscience to competitive learning." *IEEE International Conference on Neural Networks*, 1117-1124, 1988.

12. Stanley C. Ahalt. Ashok K. Krishnamurthy, Prakoon Chen. and Douglas E. Melton. "Competitive Learning Algorithms for Vector Quantization." *Neural Networks*. Vol. **3**. 277-290. 1990.

13. Stanley C. Ahalt. Ashok K. Krishnamurthy. Prakoon Chen. and Douglas E. Melton. "Performance analysis of two image vector quantization techniques." *IEEE INNS International Joint Conference on Neural Networks. Vol 1*. 169-175. 1989.

14. R. M. Gray. "Full Searched and Tree Searched Vector Quantization." *Proceedings of 1982 ICASSP*. 593-596. Paris. Apr. 1982.

15. Biing-Hwang Juang. " Multiple Stage Vector Quantization for Speech Coding." *Proceedings of 1982 ICASSP*. 597-600. Paris, Apr. 1982.

16. A. Gersho and B. Ramamurthi. "Image coding using vector quantization." *Proceedings of 1982 ICASSP*. 428-431. Paris. Apr. 1982.

INITIAL DISTRIBUTION LIST

		No. of Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Professor Murali Tummala, Code EC/Tu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	5
5.	Professor Charles Therrien, Code EC/Ti Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
6.	Dr. R. Madan (Code 1114SE) Office of Naval Research 800 North Quincy Street Arlington, Virginia 22217-5000	1
7.	Mr. John Hager (Code 70E1) Naval Undersea Warfare Engineering Station Keyport, Washington 98345	1

8.	LT Bruce E. Watkins 889 Jewell Ave. Pacific Grove, California 93950	1